Aplikacje WWW. Wykład #7

Bezpieczeństwo (i niebezpieczeństwo) w aplikacjach webowych

1. Atak typu Distributed Denial of Service (DDoS)

Atak **DDoS (Distributed Denial of Service)** to cyberatak, który ma na celu przeciążenie serwera lub sieci ogromną ilością fałszywego ruchu z wielu źródeł, aby zablokować dostęp do usług dla zwykłych użytkowników. Atak ten wykorzystuje sieć zainfekowanych komputerów i urządzeń (botnet) i może prowadzić do spowolnienia, a nawet całkowitego wyłączenia atakowanej usługi, powodując straty finansowe i wizerunkowe dla ofiary.

Nie jest to atak typowy tylko dla aplikacji webowych, ale również dla wszelkiego rodzaju usług sieciowych.

Jak działa atak DDoS?

- **Sieć botnet:** Atakujący przejmuje kontrolę nad dużą liczbą urządzeń (komputery, smartfony, kamery IP), tworząc sieć zainfekowanych maszyn zwanych botnetem.
- **Synchronizacja:** W zsynchronizowanym momencie wszystkie zainfekowane urządzenia wysyłają jednocześnie żądania do celu ataku, zasypując go fałszywym ruchem.
- **Przeciążenie:** Atakowany serwer lub sieć nie jest w stanie obsłużyć tak dużej liczby żądań, co prowadzi do wyczerpania jego zasobów (np. mocy obliczeniowej, pasma sieciowego).
- **Skutek:** Zwykli użytkownicy nie mogą uzyskać dostępu do usługi, ponieważ staje się ona niedostępna lub działa bardzo wolno.

Cele i motywy ataków DDoS

- **Konkurencja:** Atak może być przeprowadzony przez konkurencję, aby zaszkodzić reputacji i biznesowi ofiary, szczególnie w okresach wzmożonego ruchu.
- **Motywacje polityczne lub ideologiczne:** Ataki mogą być przeprowadzane przez grupy haktywistyczne lub aktywistów w ramach protestu przeciwko działaniom rządu lub firm.
- **Wyłudzenie środków finansowych:** Atak DDoS może być wykorzystywany jako narzędzie do szantażu lub próby wyłudzenia pieniędzy. Odwrócenie uwagi: Atak może być przeprowadzony jako dywersja, aby odwrócić uwagę od innej, bardziej złożonej operacji, takiej jak kradzież danych.

Rodzaje ataków DDoS

- **Ataki wolumetryczne:** Mają na celu "zalanie" celu ogromną ilością danych, wyczerpując dostępne pasmo sieciowe.
- **Ataki na protokoły:** Wykorzystują luki w protokołach sieciowych, aby przeciążyć serwer.
- **Ataki na warstwę aplikacji:** Skierowane są na konkretne aplikacje lub usługi, próbując wykorzystać ich słabości, aby zablokować ich działanie.

Jaka jest różnica między atakiem DoS a DDos?

Główna różnica jest taka, że atak typu DoS pochodzi z jednej maszyny, a nie z sieci maszyn (najczęściej botnetu).

2. Podatność Cross Site Scripting (XSS)

Podatność XSS jest jedną z najczęściej wykorzystywanych metod do przeprowadzania ataków na serwisy WWW. Polega ona na możliwości osadzenia kodu pochodzącego z zewnątrz np. poprzez formularz osadzony na atakowanej stronie, który nie jest zabezpieczony w odpowiedni sposób. Taki kod może być przekazany przez dowolne żądanie protokołu HTTP akceptowane po stronie serwera, również w postaci spreparowanego linku.

Ten rodzaj ataku można podzielić na kilka rodzajów.

Reflected XSS polega na "odbiciu" przez serwer treści przekazanej przez użytkownika/atakującego bez jego weryfikacji i wykonanie na stronie serwera lub klienta (np. JavaScript).

Inny rodzaj to **stored XSS lub persistent XSS**, który polega na zapisaniu złośliwego kodu w bazie danych lub systemie zewnętrznym licząc na to, że użytkownik odwiedzi podstronę, która ten kod z bazy wczyta i uruchomi. Wszędzie tam gdzie użytkownicy mogą tworzyć treść (np. blog, komunikator) taka podatność może wystąpić. Jednym ze sposobów na obronę jest wykorzystanie oprogramowania typu Web Application Firewall (WAF).

Istnieje kolejny typ ataku XSS, który nosi nazwę **DOM XSS**, a polega on na wstrzyknięciu złośliwego kodu bez wysyłania żądania do serwera, więc tylko po stronie przeglądarki klienta z wykorzystaniem języka JavaScript. Może się tak zdarzyć, jeżeli w źródle strony osadzony jest kod JavaScript, który wykorzystuje takie funkcje jak `eval`, `document.write` czy `innerHTML`, do których kod można wstrzyknąć np. poprzez wykorzystanie zdarzenia `hashchange` w połączeniu z przekazywaniem wartości poprzez treść żądania.

Zdarzenie `hashchange` działa tak, że nasłuchuje zmiany wartości w polu URL przeglądarki po znaku `#` (stąd jego nazwa) i w momencie jego zmiany wykonuje kod osadzony na tym evencie (takie nazwy noszą zdarzenia w języku JavaScript).

Skutki wykorzystania takiego ataku są już do przewidzenia. Można wykorzystać możliwość wydobycia danych, których twórca aplikacji na pewno nie chciał udostępniać, takich jak klucz api umożliwiający wykonanie dowolnej akcji w systemie, którą mógłby wykonać użytkownik posiadający ten klucz API. Dzięki odpowiednio spreparowanemu kodowi JavaScript możliwe jest zarówno odczytanie danych sesji, jak i ich wysłanie do zewnętrznego hosta - tu zapewne atakującego w celu przechwycenia tych danych.

Możliwe jest też poznanie działania elementów takiego serwisu i spreparowanie linku, podesłanie go do użytkownika, który jeżeli bezmyślnie w niego kliknie, pozwoli wykonać akcję w kontekście zalogowanego użytkownika, np. kliknięcie (poprzez kod JavaScript) odkrytego przycisku do usuwania ważnych danych.

Ujmując rzecz ogólnie, podatność XSS pozwala na wykonanie dowolnego kodu JavaScript, który ma dzisiaj dość duże możliwości.

Podstawowe metody obrony (krótko):

- Automatyczne zamienianie znaków specjanych na encje na wszystkich warstwach (HTML, JS, CSS, URL).
- Używanie bezpiecznych API (textContent zamiast innerHTML), unikanie `eval` i `document.write`.
- **Content Security Policy (CSP)**, HttpOnly i Secure dla ciasteczek.
- Walidacja i dezynfekcja (sanitizacja) danych po stronie serwera oraz biblioteki sanitizujące dla HTML.
- Wykorzystywanie systemu szablonów, który automatycznie zamienia wszystkie potencjalnie nieporządane znaki na encje i uniemożliwia zmianę treści szablonów z poziomu użytkownika.
- Regularne testy bezpieczeństwa i przegląd ścieżek wejścia.

Czym jest dezynfekcja danych?

Jest to proces usuwania nieporządanych jego części, w tym kontekście elementów dokumentu HTML, które uznane są za potencjalnie niebezpieczne.

Więcej: https://en.wikipedia.org/wiki/HTML_sanitization

Na czym polega Content Security Policy (CSP)?

Content Security Policy (CSP) to mechanizm bezpieczeństwa, który chroni strony internetowe przed niektóre typami ataków, takimi jak ataki XSS, poprzez kontrolę zasobów, które przeglądarka może ładować i wykonywać. Działa on na podstawie zestawu reguł zdefiniowanych przez administratora strony, które określają, skąd mogą pochodzić poszczególne zasoby (np. skrypty, arkusze stylów, obrazy). CSP jest implementowane poprzez ustawienie nagłówka Content-Security-Policy w odpowiedziach HTTP z serwera.

- **Określanie dozwolonych źródeł:** CSP pozwala określić, z jakich domen przeglądarka może pobierać np. skrypty, arkusze stylów czy obrazy, blokując wszystkie inne. Na przykład, dyrektywa script-src może zezwalać na ładowanie skryptów tylko z konkretnych, zaufanych domen, jak w przykładzie: script-src static.strona.tld https://inna-strona.tld;.
- **Blokowanie nieautoryzowanych zasobów:** Jeśli zasób (np. skrypt) próbuje zostać załadowany ze źródła, które nie jest wymienione w polityce, przeglądarka go nie załaduje. Nawet jeśli złośliwy kod zostanie wstrzyknięty na stronę, CSP może uniemożliwić jego wykonanie, jeśli nie pochodzi z dozwolonego źródła.
- **Ochrona przed clickjackingiem:** Można skonfigurować CSP, aby zapobiec osadzaniu strony w ramkach na innych witrynach, używając dyrektywy frame-ancestors 'none', co chroni przed atakami clickjacking.
- **Tryb raportowania:** Istnieje również tryb raportowania (Content-Security-Policy-Report-Only), który nie blokuje zasobów, ale wysyła raporty o naruszeniach. Jest to przydatne do testowania polityki przed jej pełnym wdrożeniem.

Więcej na temat XSS:

- https://sekurak.pl/czym-jest-xss/
- https://sekurak.pl/do-czego-mozna-wykorzystac-xss-czyli-czym-jest-beef/
- https://owasp.org/www-community/attacks/xss/

3. Podatność Cross-Site Request Forgery (CSRF)

Ten rodzaj ataku nazywany jest też czasem atakiem XSRF, Sea Surf, Session Riding czy one-click attack.

Jest to atak, który odbywa się po stronie przeglądarki i polega na wykonaniu akcji (żądania HTTP) przez zalogowanego użytkownika, która wykonywana jest przez niego nieświadomie, nie była zamierzona.

Przykład 1 (za "Bezpieczeństwo Aplikacji Webowych" strona 374, Securitum, Wydanie I poprawione, Kraków 2021)

1. Użytkownik posiadający konto w serwisie forum umieszcza komentarz, którego część może wyglądać tak: `<img src="http://forum.jakas.domena.pl/admin/user/add?login=newadmin&pass=1234&type=adm">` i jest on następnie zapisywany do bazy danych. Oczywiście zakładamy, że po drodze nie odbywa się dezynfekcja danych, która mogłaby ten problem

zneutralizować. Nie byłoby to jednak takie proste, gdyż musiałoby to się odbyć poprzez analizę wartości atrybutu `src`.
2. Administrator loguje się do panelu i wyświetla listę komentarzy w celu moderacji.
3. Dane ładowane są z bazy, a przeglądarka podczas próby rendorowania obrazu wysyła nieautoryzowane żądanie typu `GET` do serwera. Należy również zwrócić uwagę, że ten rodzaj ataku wymaga, aby atakujący znał nieco architekturę atakowanego systemu.

Atak odbył się z komputera administratora, więc w logach będzie widoczny jego adres IP, a nie atakującego. Atak nastąpił z tej samej domeny (używa się czasem nazwy takiego ataku jako On-site Request Forgery OSRF), więc to również mogłoby nie zostać zablokowane np. przez CORS.

Innym atakiem, który można przeprowadzić poprzez podatność CSRF jest atak z wykorzystaniem żądania HTTP typu POST oraz różnych domen.

1. Użytkownik loguje się do np. bankowości internetowej.
2. Atakujący ma przygotowaną domenę ze stroną zawierającą kod z formularzem, który zawiera pola pozwalające na wykonanie akcji w tym samym serwisie co użytkownik, np. wysłanie przelewu, ale formularz jest wysyłany na adres serwisu, z którego korzysta użytkownik bankowości internetowej.
3. Do osiągnięcia celu wymagane jest, aby w trakcie trwania sesji użytkownik odwiedził stronę, której kod natychmiast wyśle spreparowany formularz.

Jest to oczywiście dość naiwny przykład w dzisiejszych czasach, gdyż musi zajść wiele czynników jednocześnie:

- nieuwaga użytkownika,
- brak zabezpieczenia wykonywania żądania między domenami, które nie są wzajemnie zaufane,
- brak potwierdzenia realizacji zlecenia,
- brak zabezpieczeń CSRF.

Oczywiście większość dzisiejszych serwisów posiada już zabezpieczenia, które takie ataki skutecznie blokują.

Jako ciekawostkę można tutaj podać magiczny parametr formularza o nazwie `_method`, który umożliwia wywołanie innych niż `GET, POST` żądań za pomocą formularza. Zachęcam do samodzielnego sprawdzenia.

Wśród innych podatności typu CSRF można również wymienić możliwość wywołania żądania, które wykorzystuje metodę uwierzytelniania typu `HTTP Basic Authentication` i wysyła je na urządzenie w sieci lokalnej.

Przykład:

```
<img src="http://admin:admin@192.168.1.1/reboot/">
```

Ochrona przez CSRF

1. Zabezpieczenie miejsc aplikacji, które umożliwiają zmianę ważnych parametrów systemu.

2. Losowe tokeny. Jest to zalecana przez OWASP metoda (Synchronized Token Pattern), która polega na generowaniu pseudolosowego tokena po stronie serwera i przekazywania go do klienta, a następnie osadzanie w formularzu i weryfikacja po stronie serwera, czy token jest prawidłowy i przypisany do

danego użytkownika. Atakujący nie znając tokena, nie może przygotować działającego formularza, ale wyciek tokena pozwala obejść to zabezpieczenie.

3. Zmiana wszystkich żądań typu `GET`, które zmieniają stan aplikacji, na żądania typu `POST`. Chronimy w ten sposób część interfejsu aplikacji poprzez ukrycie parametrów żądania (`GET` przekazuje je przez URL, więc są widoczne i w logach, ale również np. poprzez nagłówek `Referer` (a nie `Referrer` - patrz materiały, ciekawostki), który może doprowadzić do wycieku tokena). Możliwe jest również uzyskanie dostępu do tokena CSRF poprzez podatność XSS, która może go zarówno odczytać jak i przygotować spreparowane żądanie i je wysłać.

4. Korzystanie z rozwiązań np. frameworków, dodatkowych bibliotek, które dostarczają zebezpieczenia podatności CSRF. Należy jednak dobrze to zweryfikować.

5. Atrybut `SameSite` w ciasteczkach. Takie rozwiązanie powoduje wysłanie żądania do aplikacji, ale jest traktowane jako nieuwierzytelnione i nie zmienia stanu aplikacji.

6. Należy również zabezpieczyć formularz logowania przed atakiem tego typu. Istniała historycznie podatność w serwisie Google, która polegała na przekazaniu przez (ponownie) podstawioną stronę z automatycznie wysyłającym się żądaniem uwierzytelnienia danymi włamywacza (nowe konto), a następnie wszystkie odwiedzane adresy (jeżeli ofiara się nie zorientowała, że nie jest zalogowana na swoim koncie) były zapisywane w historii konta Google, gdzie atakujący mógł szukać takich, które warto zaatakować.

Warto tu zaznaczyć, że zagrożenia typu CSRF nie są już aktualnie na liście najczęściej przeprowadzanych ataków na aplikacje webowe wg. OWASP Top10.

Więcej na temat CSRF:

- https://owasp.org/www-community/attacks/csrf

4. Podatność SQL Injection

Definicja tej podatności za Wikipedia (https://pl.wikipedia.org/wiki/SQL_injection):

> "SQL injection (z ang.) – metoda cyberataku wykorzystująca lukę w zabezpieczeniach aplikacji polegającą na nieodpowiednim filtrowaniu lub niedostatecznym typowaniu danych użytkownika, które to dane są później wykorzystywane przy wykonaniu zapytań (SQL) do bazy danych. Podatne są na nią wszystkie systemy przyjmujące dane od użytkownika i dynamicznie generujące zapytania do bazy danych"

Wiadomo, że jeżeli pozwolimy użytkownikowi na przekazywanie dowolnych zapytań typu `SELECT` SQL do naszej bazy danych to nie możemy mówić o istnieniu zabezpieczeń. Nawet jeżeli możliwości użytkownika kończą się na możliwości przekazania wartości pola do zapytania `SELECT` polegającego na filtrowaniu, wciąż możliwe jest przekazanie fragmentów, które zupełnie zmieniając sposób działania całego zapytania. Poniżej przykład bazujący na przykładzie z Wikipedii.

Przykład 1:

```
# kod prezentujący możliwy wektor ataku z podatnością SQL injection
```

```
# user_id pobrane z formularza albo np. z URL
# /users?user_id=1
user_id = '1'
query = f"SELECT * FROM users WHERE USER_ID={user_id};"
print(query)

# a jeżeli przekażemy wartość np. tak
# /users?user_id=1 OR 1=1
user_id = '1 OR 1=1'
query = f"SELECT * FROM users WHERE USER_ID={user_id};"
print(query)

# wyjście
SELECT * FROM users WHERE USER_ID=1;
SELECT * FROM users WHERE USER_ID=1 OR 1=1;
```

Bez odpowiedniego mechanizmu np. kontroli typu, łatwo jest wykorzystać taką podatność, mimo tego, że dane tylko jednego pola są wprowadzane np. przez formularz. W tym przypadku drugie zapytanie zwróci wszystkie dane z tabeli users, co można z powodzeniem nazwać wyciekiem danych.

Oczywiście atakujący nie wie jak wygląda faktyczne zapytanie uruchamiane po stronie serwera, więc początkowy etap polega na sprawdzeniu czy podatność występuje.

Można więc dodatkowo dodać do wstrzykiwanego zapytania fragment, który zakomentuje ewentualną dalszą część zapytania. Jest to też przykład "ucieczki" ze stringa samego zapytania, aby dodać swój własny kod zapytania.

Przykład 2:

```
# możliwy adres URL aplikacji
/search?phrase=szukam'--

# możliwa postać zapytania SQL
SELECT * FROM posts WHERE content LIKE '%szukam'--%' AND published = 1
```

Część zapytania za znakami `--` zostanie potraktowana jako komentarz, więc nie zostanie wykonana. To już wykonuje zapytanie niezgodnie z przeznaczeniem, czyli zwraca wszystkie posty, ktorych treść kończy się słowem `test`, ale może być jeszcze gorzej, jeżeli dodamy fragment polecenia zaprezentowanego w przykładzie numer 1.

Przykład 3:

```
# możliwy adres URL aplikacji
/search?phrase=szukam' OR 1=1--

# możliwa postać zapytania SQL
SELECT * FROM posts WHERE content LIKE '%szukam' OR 1=1--%' AND published = 1
```

Teraz zwrócone zostaną wszystkie rekordy z tabeli `posts` gdyż warunek `1=1` jest zawsze prawdziwy!

Wykorzystanie SQL injection

Wykorzystanie sposobu UNION-based

Jest to sposób, który polega na wstrzykiwaniu (poprzez doklejanie/konkatenację łańcuchów zapytania) do istniejącego zapytania zapytania rozpoczynającego się od `UNION`, które ma wydobyć dane zupełnie nie związane z oryginalnym zapytaniem.

Główną kwestią do rozwiązania z poziomu atakującego jest zdobycie wiedzy na temat liczby kolumn, które oryginalne zapytanie zwraca. W przypadku silników baz danych Oracle, SQL Server, PostgreSQL muszą zgadzać się również typy danych, ale w przypadku MySQL/MariaDB już nie.

Najpopularniejszym sposobem jest stopniowe odpytywanie poprzez zwiększanie liczby kolumn, które zwraca wstrzyknięte zapytanie UNION. Nie wiem zapewne na początka jakie typy danych mogą być określone dla każdej z kolumn, ale to można na razie obejść poprzez wykorzystanie wartość `null` w zapytaniu.

Przykład 4:

```
# możliwy adres URL aplikacji
/search?phrase=szukam' UNION SELECT null--

# możliwa postać zapytania SQL
SELECT * FROM posts WHERE content LIKE '%szukam' UNION SELECT null--
```

Kolejne próby polegają na dodawaniu kolejnych wartości `null`, aż zapytanie wykona się poprawnie i dowiemy się ile kolumn znajduje się w zapytaniu oryginalnym.

Innym sposobem na zdobycie wiedzy o liczbie kolumn jest wykorzystanie klauzuli `ORDER BY` oraz możliwości określenia numeru kolumny, po której następuje sortowanie, wykorzystując zasadę "dziel i zwyciężaj".

Jak zdobędziemy już wiedzę na temat liczby kolumn, to można przystąpić do budowy docelowego zapytania, którego celem jest wydobycie cennych dla atakującego informacji.

Takie zapytanie mogloby wyglądać tak:

Przykład 5:

```
/search?phrase=szukam' UNION SELECT null, null, table_name FROM
information_schema.tables--

SELECT * FROM posts WHERE content LIKE '%szukam' UNION SELECT null, null,
table_name FROM information_schema.tables--
```

I tu widzimy już jakie możliwości daje taka podatność. Dalej możemy już włączyć funkcje konkatenacji po stronie silnika bazy danych, aby wybierać więcej danych zachowując porządaną liczbę kolumn w zapytaniu

`UNION` i wyciągnąć z bazy wszystkie dane, do których dostęp ma użytkownik bazodanowy wywołujący to zapytanie.

Ograniczeniami takiego podejścia jest ograniczenie ataku tylko do zapytań typu `SELECT` oraz fakt, że jeżeli zapytanie posiada ciąg dalszy za wstrzykiwanym fragmentem z `UNION` to całość nie zadziała, pojawi się błąd składniowy.

Można więc spróbować wykorzystać inne podejście.

Wykorzystanie sposobu ERROR-based

Tu należy zaznaczyć, że sposób jest możliwy do wykorzystania pod warunkiem, że do klienta zwracana jest dokładna treść błędu po stronie bazy danych to można nim w pewnym zakresie sterować.

Jako przykład można wykorzystać wbudowaną funkcję, która odbija do komunikatu błędu wartość do niej przekazaną (czyli np. zwróconą wartość zmiennej, czy poprawnego zapytania SQL), np. funkcja `CAST` dla PostgreSQL.

Przykład 6:

```
# PostgreSQL
SELECT cast(version()) as integer);

# przykładowe wyjście
error: invalid input syntax for integer: "PostgreSQL 10.0 on x86_64-pc-linux-gnu
... "

# MySQL/MariaDB
select extractvalue(1, concat('!', version()));

# przykładowe wyjście
Error Code: 1105. XPATH syntax error: '!8.0.22'
```

Dalsza faza polega na przekazywaniu do tych funkcji zapytań, które faktycznie mają wyciągnąć jakieś przydatne dla atakującego dane, wykorzystując między innymi konkatenację, aby do funkcji przekazana została porządana liczba argumentów (tu 1).

Większość popularnych silników SQL jest podatna na tego typu atak, jeżeli nie została poprawnie skonfigurowana w kontekście obsługi komunikatów o błędach.

Atak na ślepo (ang. BLIND SQL injection)

Tutaj można wyróżnić dwa podejścia:

- atak na ślepo oparty o treść
- atak na ślepo oparty o czas

Atak na ślepo oparty o content polega na wykorzystaniu własności zapytania SQL, które po wstrzyknięciu dodatkowego warunku np. `OR 1=1` lub `OR 1=2` pozwoli nam na sprawdzenie czy oryginalne zapytanie zwraca

prawdę czy fałsz (ale to już będziemy wiedzieć z "normalnej" pracy aplikacji), ale przede wszystkim możemy dowiedzieć się, jaką wartość logiczną zwraca wstrzykiwany fragment.

Często przytaczanym przykładem jest wykorzystanie funkcji typu `SUBSTRING`, które mogą być wykorzystane do wydobywania informacji o np. schemacie bazy danych w pierwszym etapie, a później o dowolnych danych w kolejnych krokach. Takie podejście wymaga jednak wielu prób, żądań i czasu, więc mimo swojej uniwersalności może zostać jednak zablokowane np. przez mechanizmy dławiące.

Przykład 7:

```
<oryginalne zapytanie> AND substring(name, 1, 1) = 'a'
...
# można wykorzystać również kody znaków ASCII, co powoli na wdrożenie
# strategii dziel i zwyciężaj
<oryginalne zapytanie> and ascii(AND substring(name, 1, 1)) < 128
# itd.
```

Atak na ślepo oparty o czas ma podobny cel, stworzyć zapytanie logiczne, które serią pytań zamkniętych (Prawda lub Fałsz) pozwoli nam na zdobycie odpowiedzi na nurtujące nas pytania, np. jaką nazwę nosi dana tabela (poczynając od np. od kolejnych liter), później jakie kolumny w niej występują, a finalnie już same dane.

Sposób polega na wykorzystaniu wbudowanych funkcji pozwlających na wstrzymanie wykonania zapytania o określony czas np. funkcja `SLEEP()` w silniku MySQL/MariaDB oraz wdrożonego sposobu rozwiązywania wyrażeń logicznych, który zazwyczaj bazuje na `short-circuit evaluation`. Obserwując więc czas wykonania zapytania może stwierdzić, czy osadzony warunek jest prawdziwy czy nie.

Przykład 8:

```
<oryginalne zapytanie> AND 1=1 AND SLEEP(3) = 1
<oryginalne zapytanie> AND 1=2 AND SLEEP(3) = 1
```

Jeżeli w miejsce warunków `1=1` oraz `1=2` wstawimy dowolny warunek, którego sprawdzenie nas interesuje to w zależności od logiki całego warunku albo jego ostatnia część wykorzystująca funkcję `SLEEP` będzie ewaluowana albo nie, co będzie miało bezpośredni wpływ na wydłużenie czasu wykonania zapytania.

Wiele zapytań przy jednym wywołaniu (stacked queries)

Inną możliwością wykorzystania podatności SQL injection jest umożliwienie wykonania wielu niezależnych zapytań przy jednym wywołaniu. Jeżeli wiemy już, że można wstrzykiwać zapytania to może być możliwe wykonanie chociażby poniższego polecenia SQL.

Przykład 9:

```
/search?phrase=szukam'; DROP database blog--

SELECT * FROM posts WHERE content LIKE '%szukam'; DROP database blog--
```

Dzięki tej podatności możliwe jest wykonanie ataków opisanych poniżej.

Wydobycie dowolnych danych z bazy (dane osobowe, dane logowania, dane medyczne, pytania pomocnicze w przypadku zagubienia hasła, itp.).

Ominięcie ekranu logowania.
Jeżeli w systemie zaimplementowano dość naiwną metodę sprawdzenia czy poświadcznienia logowania (nazwa użytkownika, hasło) zwracają jakiś wynik (zliczanie) to wstrzykując polecenie SQL zgodnie z wcześniejszymi zasadami, czyli np. preprarując takie zapytanie `SELECT * FROM users WHERE login='admin' OR 1=1'-- AND password='adminadmin'`

Jeżeli jednak w systemie wykorzystywany jest mechanizm, który polega na pobraniu hasha hasła z bazy i porównanie go na poziomie backendu aplikacji to wciąż można przeprowadzić atak.

Zakładając, że zapytanie wybierające hash hasła z bazy ma postać:

```
# dane w formularzu
# username: admin
# password: adminadmin

SELECT password_hash FROM users WHERE username='admin';
```

możemy spróbować je spreparować w poniższy sposób, przy założeniu, że wiemy lub sprawdziliśmy metodą prób i błędów, że algorytm hashujący to SHA1:

```
# dane w formularzu
# username: admin' AND 1=2 UNION SELECT
'dd94709528bb1c83d08f3088d4043f4742891f4f'--
# password: adminadmin
# wyliczony przez nas hash hasła adminadmin:
dd94709528bb1c83d08f3088d4043f4742891f4f

SELECT password_hash FROM users WHERE username='admin' AND 1=2 UNION SELECT
'dd94709528bb1c83d08f3088d4043f4742891f4f'--;
```

Analizując zapytanie dojdziemy do wniosku, że warunkek `username='admin' AND 1=2` jest zawsze fałszywy, więc zapytanie zwróci wstrzyknięty przez nas hash, a ten będzie równy hashowi hasła podanego w formularzu co będzie logicznie prawdziwe i zostaniemy zalogowani do aplikacji!

Poprzez podatność **SQL injection stacked queries** możliwe są jeszcze takie operacje jak:

- modyfikacja danych w bazie np. aktualizacja hasła dowolnego użytkownika,
- odczyt dowolnego pliku z serwera bazy danych np. poprzez przechowywanie ścieżek do plików w niezmienionej formie w bazie danych, co pozwala np. na wykorzystanie podatności o nazwie **path traversal** poprzez wstawienie chociażby takiej ścieżki jak '../../../../../../../etc/passwd' pod wybrany

przez siebie wpis w tabeli z plikami, a następnie odwołanie się żądaniem `GET` do tego pliku i zwróceniem jego zawartości do przeglądarki,
- wykonanie kodu, jeżeli korzystamy np. z systemu CMS, który umożliwia przechowywanie np. szablonów czy innych elementów w postaci kodu źródłowego do wykonania,
- odczyt plików z dysku poprzez wbudowane funkcje silników SQL pozwalające na ich odczyt, np. `load_file` w MySQL,
- zapis plików na dysku serwera,
- wykonanie poleceń systemu operacyjnego.

Jak widać ta podatność jest bardzo niebezpieczna i bez wykorzystania odpowiednich mechanizmów trudna do wykrycia bez dogłębnej analizy.

Obrona przed SQL injection

1. Zapytania parametryzowane.

Zapytania parametryzowane polegają na stworzeniu szablonu zapytania z wyróżnieniem pól, które zostaną do takiego zapytania wstawione w miejsce specjalnie zdefiniowanych zmiennych. Przykład takiego zapytania z wykorzystaniem języka PHP oraz fraemworka Doctrine:

źródło: https://www.doctrine-project.org/projects/doctrine-dbal/en/4.3/reference/data-retrieval-and-manipulation.html

```
<?php
# wykorzystując funkcję do zapisania wartości zmiennej jako łańcuch znaków
# który ograniczy możliwości SQL injection
$sql = "SELECT * FROM articles WHERE id = '" . mysql_real_escape_string($id,
$link) . "'";
$rs = mysql_query($sql);

# albo z zapytaniem parametrycznym
// $conn instanceof Doctrine\DBAL\Connection
$sql = "SELECT * FROM articles WHERE id = ? AND status = ?";
$stmt = $conn->prepare($sql);
$stmt->bindValue(1, $id);
$stmt->bindValue(2, $status);
$resultSet = $stmt->executeQuery();

# można też określić typ wartości
$date = new \DateTime("2011-03-05 14:00:21");
$stmt = $conn->prepare("SELECT * FROM articles WHERE publish_date > ?");
$stmt->bindValue(1, $date, "datetime");
$resultSet = $stmt->executeQuery();
```

W przypadku języka Python wykorzystanie interpolacji stringów również pozwala na wykorzystanie podeobnego mechanizmu:

```
query = "UPDATE BookList SET Book = %s WHERE Book = %s"
cursor.execute(query, ("Changing Title", "A Really Good Book"))
```

2. Walidacja typów danych

Korzystając z rozwiązań SQL, gdzie potrzebujemy wykorzystać własne polecenia jednym ze sposobów zabezpieczenia takich zapytań jest sprawdzanie typów dla wartości przekazywanych przez użytkownika. W przypadku typów łańcuchowych sytuacja jest nieco trudniejsza, gdyż tutaj powinniśmy również wykonać sprawdzenie zgodności formatu jeżeli da się go określić. Można też wykorzystać wyrażenia regularne, które mogą wyłapać najbardziej popularne sposoby ataku poprzez podatność SQL injection.

3. Wykorzystanie bibliotek typu ORM

Wykorzystanie biblioteki ORM, która posiada już zaimplementowane zabezpieczenia przed większością podatności SQL injection będzie dobrym wyborem.

Poniżej przykład wykorzystania SQLAlchemy.

```
userToUpdate = session.query(Users).filter(Users.Name == "admin").one()
userToUpdate.Name = "administrator"
session.commit()
```

4. Hardening bazy danych

Pojęcie "hardeningu" to proces wzmacniania zabezpieczeń systemów informatycznych poprzez usuwanie zbędnych funkcji, usług i aplikacji oraz konfigurowanie pozostałych elementów w celu zminimalizowania podatności na ataki.

W przypadku baz danych może to być:

- użytkownik bazy danych, który został wykorzystany do wykonywania zapytań nie powinien być użytkownikiem administracyjnym,
- w miarę możliwości można stosować separację użytkowników na poziomie bazy danych, tak aby konto zwykłego użytkownika nie miało dostępu do tabel, do których dostęp powiniem mieć tylko administrator,
- wyłączenie obsługi `stacked queries`,
- wyłączenie potencjalnie niebezpiecznych funkcji silnika baz danych, np. wczytywania plików, wywoływania poleceń powłoki o ile nie jest to niezbędne dla prawidłowej pracy aplikacji,
- proces bazy danych na serwerze powinien pracować z innymi niż `root` uprawnieniami,
- wdrożenie mechanizmu logowania i monitorowania,
- wykonywanie audytów przed wdrożeniem aplikacji.

Inne przykłady do przeanalizowania

- https://www.w3schools.com/sql/sql_injection.asp
- https://owasp.org/www-community/attacks/Blind_SQL_Injection

5. Inne zagrożenia dla aplikacji webowych

Zagrożeń, które umożliwiają wykonanie nieporządanych operacji w aplikacji webowej jest dużo więcej, ale nie ma tu miejsca, aby opisać je wszystkie w szczegółach.

Wyminię więc kilka innych, wybranych zagrożeń i krótko wyjaśnię ich istotę.

1. Path traversal

Zagrożenie to polega na, przytoczonym już przykladzie z poprzedniego rozdziału, wykorzystaniu niewystarczających (lub braku) zabezpieczeń przed możliwością podania dowolnej ścieżki do pliku, który np. zapisany w bazie danych w postaci wyłącznie łańcucha znaków określającego jesgo ścieżkę, może być później pobrany/wyświetlony przez atakującego.

Możliwe zagrożenia tej podatności:

- odczytanie plików z serwera np. konfiguracji, hashy haseł,
- wykonanie pliku w powłoce.

Ochrona:

- walidacja danych wejściowych,
- walidacja ścieżki przy próbie odczytu pliku, np. czy pochodzi z podkatalogu aplikacji lub ścieżki z "white listy",
- możliwe jest też wdrożenie mechanizmu identyfikacji plików,
- weryfikacja i wdrożenie odpowiednich uprawnień dla procesu serwera www.

2. Zagrożenia deserializacji

Ujmując problem ogólnie, deserializacja to proces odwrotny do serializacji, czyli przekształcenia danych z formatu binarnego lub tekstowego do obiektu w pamięci programu. Podatność deserializacji występuje, gdy aplikacja deserializuje dane pochodzące z niezaufanego źródła bez odpowiedniej walidacji i kontroli.

Ta podatność dotyczy wielu języków programowania, ale krótkie jego omówienie będzie dotyczyło języka Python oraz modułu `pickle`.

W przypadku tego modułu po serializacji będziemy widzieli ciąg bajtów, który reprezentuje zserializowane obiekty. W przypadku modułu `pickle` jest to rozwiązanie, które oferuje możliwości niewielkiego języka programowania, z pewnymi instrukcjami nazywanymi `opcodami` pozwalające na odtworzenie oryginalnego obiektu.

Dzięki tym instrukcjom możliwe jest zserializowanie niemal dowolnego obiektu Pythona ale też i wykonanie dowolnej jego funkcji przy deserializacji (konkretnie opcode `REDUCE`). Możliwe jest więc przekazanie zserializowanych danych w postaci `pickle`, który przy próbie deserializacji wywołają funkcje dostępne np. w module `os.system`.

Ochrona:

- nie używanie deserializacji jeżeli dane mogą pochodzić z niezaufanych źródeł,
- użycie bezpieczniejszego formatu danych np. `json` jeżeli nie jest wymagana serializacja skomplikowanych obiektów.

3. Zagrożenia REST API

Nadpisywanie metod HTTP.

Polega na użyciu niestandardowego nagłówka HTTP, takiego jak X-HTTP-Method-Override, aby zmusić serwer do przetworzenia żądania inną metodą niż ta, którą wysłał klient (np. w sytuacji gdy klient wysyła POST, ale chce wykonać operację PUT). Serwer odczytuje wartość z nagłówka X-HTTP-Method-Override i wykonuje operację zgodną z tą metodą, zamiast standardowej metody POST. Aplikacja wysyła żądanie POST, ale dodaje nagłówek X-HTTP-Method-Override: PUT, aby zaktualizować zasób na serwerze zamiast go utworzyć.

Efektem może być ominięcie ograniczeń klienta (np. przeglądarki), który nie obsługuje wszystkich metod HTTP, lub umożliwienie użycia metod, które są blokowane przez serwer (np. PUT do tworzenia plików, DELETE do usuwania).

Należy zachować ostrożność, ponieważ niewłaściwe obsługiwanie tych nagłówków może stworzyć luki bezpieczeństwa (np. umożliwiając nieupoważnionym użytkownikom tworzenie lub usuwanie plików).

Pozostale zagrożenia są już bardziej ogólne:

- **Uszkodzone uwierzytelnianie i autoryzacja:** Pozwala na nieautoryzowany dostęp do funkcji lub danych, a także kradzież danych uwierzytelniających (np. kluczy API).
- **Błędy konfiguracji:** Nieodpowiednie konfiguracje, na przykład brak ograniczenia liczby żądań na minutę (rate limiting), mogą prowadzić do ataków DDoS.
- **Ataki injection:** Atakujący mogą wstrzykiwać złośliwy kod (np. SQL injection) w żądaniach, aby uzyskać nieuprawniony dostęp do danych lub przejąć kontrolę nad systemem.
- **Ataki DDoS:** Rozproszone ataki odmowy usługi mogą przeciążyć API, powodując jego niedostępność dla legalnych użytkowników.
- **Nadmierne udostępnianie danych:** API zwraca więcej danych, niż jest to potrzebne, co może prowadzić do wycieku poufnych informacji, nawet jeśli klient nie prosi o nie bezpośrednio.
- **Luki w szyfrowaniu:** Przechwytywanie niezaszyfrowanej komunikacji między klientem a serwerem, co pozwala na dostęp do wrażliwych danych.
- **Problemy z zarządzaniem zasobami:** Niewłaściwe zarządzanie zasobami może prowadzić do problemów z wydajnością, a także do wykorzystania luk bezpieczeństwa.
- **Interfejsy API Shadow i Zombie:** Starsze, zapomniane interfejsy API, które są wciąż dostępne, ale nie są monitorowane ani zabezpieczone, stanowią łatwy cel dla atakujących.

6. Wbudowane zabezpieczenia frameworka Django

> Lista wbudowanych zabezpieczeń i ich wyjaśnienie jest skróconą wersją pochodzącą z oficjalnej dokumentacji Django https://docs.djangoproject.com/pl/5.2/topics/security/

Zawsze dezynfekuj dane od użytkowników

I to jest rada, która powinna nam przyświecać przy tworzeniu każdej aplikacji. Nie każdy użytkownik będzie chciał korzystać z aplikacji wedle naszego optymistycznego scenariusza.

O walidacji danych z formularzy można szczegółowo przeczytać tu:
https://docs.djangoproject.com/pl/5.2/ref/forms/validation/

Zabezpieczenie przed XSS

Poprawne wykorzystanie mechanizmu szablonów dostępnego w Django powinno nas uchronić przed większością ataków typu XSS. Każda wartość jest poddawana dezynfekcji, ale są funkcje, które mogą ten mechanizm złagodzić lub wyłączyć. Warto przeczytać o wyjątkach w dokumentacji.

Zabezpieczenie przed CSRF

Zabezpieczenie przed atakami typu CSRF powinno być włączone (domyślnie jest) i jest obsługiwane między innymi poprzez `MIDDLEWARE` ale również powinno być osadzane na każdym widoku, formularzu.

Szczegóły oraz punkty obsługi CSRF w aplikacji Django są opisane tutaj: https://docs.djangoproject.com/pl/5.2/howto/csrf/#using-csrf

Zabezpieczenie przed SQL injection

Django ORM wykorzystuje zapytania sparametryzowane, więc polecenie jest oddzielone od jego parametrów i dodatkowo poddawane formatowaniu jako tekst (ang. escaping).

Django ORM pozwala jednak na osadzanie własnych, surowych zapytań SQL i tam musimy ustrzegać się wszystkich zagrożeń, które zostały opisane wcześniej.

Ochrona przed zagrożeniami typu `Clickjacking`

Jest to zabezpiecznie, które chroni stronę przez osadzeniem jej wewnątrz znacznika `<iframe>` i wykorzystania jej w nieporządany sposób. Domyślnie to zabezpieczenie jest włączone.

Więcej można doczytać tutaj: https://docs.djangoproject.com/pl/5.2/ref/clickjacking/#clickjacking-prevention

SSL/HTTPS

Django obsługuje wykorzystanie protokołu HTTPS, który de facto jest obligatoryjny przy wykorzystaniu z jego wersją 2.0, poprzez szereg opcji uniemożliwiających komunikację po HTTP i automatyczne przekierowania na HTTPS. Obsługuje również HSTS (HTTP Strict Transport Security). Więcej w dokumentacji.

Walidacja nagłówków hosta

Django umożliwia zdefiniowanie dopuszczalnych adresów hostów, które pojawiają się w nagłówkach żądań poprzez zmienną `ALLOWED_HOSTS` w pliku `settings.py`, która to domyślnie jest pusta. W środowisku produkcyjnym należy to zdefiniować zgodnie z konfiguracją wirtualnych hostów np. w serwerze Apache, który może być naszym docelowym rozwiązaniem.

To powinno nas chronić przed podatnościami typu CSRF (automatyczne dezynfekcja wartości nagłówków powinna zaś chronić przed XSS) poprzez atak na nagłówek.

O możliwych atakach na nagłówki HTTP można poczytać między innymi tu: https://portswigger.net/web-security/host-header

Referrer policy

Django pozwala na zdefiniowanie polityki ustawiania wartości `Refferer`, która określa skąd użytkownik pojawił się na danej podstronie. Zostało to już wspomniane przy okazji podatności CSRF.

Cross-origin opener policy

Ten mechanizm chroni przed atakami typu cross-origin. Więcej:
https://docs.djangoproject.com/pl/5.2/ref/middleware/#cross-origin-opener-policy

Session security

Tutaj jednym z zagrożeń jest polityka cross-origin, która umożliwia definiowanie ciasteczek dla całej domeny, ale jeżeli możliwe jest przejęcie kontroli nad subdomeną, to możliwe jest też przeprowadzenie stosownego ataku. Mechanizm sesji w Django ma swoje ograniczenia, warto o nich przeczytać w dokumentacji:
https://docs.djangoproject.com/pl/5.2/topics/http/sessions/#session-security

Zawartość uploadowana przez użytkownika

Zagrożenia z tym związane zostały już opisane przy okazji kilku opisywanych podatności. Dobrym pomysłem jest delegacja serwowania plików statycznych na serwisy typu CDN. Tutaj obowiązują również zdroworozsądkowe zasady zarządznia zasobami plikowymi:

- definiowanie maksymalnych rozmiarów plików,
- dławienie ilości przesyłanych plików, przepustowości,
- odpowiednie uprawnienia (np. brak możliwości wykonania),
- filtrowanie typów przesyłanych plików,
- filtrowanie ścieżek,

Inne zagrożenia

Tu już zacytuję oryginał:

> Make sure that your Python code is outside of the web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).

> Take care with any user uploaded files.

> Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or web server module to throttle these requests.

> Keep your SECRET_KEY, and SECRET_KEY_FALLBACKS if in use, secret.

> It is a good idea to limit the accessibility of your caching system and database using a firewall.

> Take a look at the Open Web Application Security Project (OWASP) Top 10 list which identifies some common vulnerabilities in web applications. While Django has tools to address some of the issues, other issues must be accounted for in the design of your project.

> Mozilla discusses various topics regarding web security. Their pages also include security principles that apply to any system.

7. Podsumowanie

Mnogość możliwości przeprowadzenia ataków i podatności może być przytłaczająca. Zwłaszcza dla osób, które chciały tylko stworzyć prostą stronę osobistą, uruchomić blog. Odwieczna zasada "system jest tak bezpieczny jak jego najsłabszy punkt" działa również w tym przypadku. Jedna podatność może uruchomić całą kaskadę zdarzeń i zniweczyć pracę włożoną w inne zabezpieczenia. Warto więc mieć ich świadomość, ale też świadomość, że audytowanie i monitorowanie może znacznie poprawić to bezpieczeństwo.

7. Źródła wiedzy, materiały, narzędzia

1. Projekt OWASP oraz aktualna lista TOP 10 zagrożeń wg. tej organizacji: https://owasp.org/Top10/
2. Web security wg. Mozilla Security Assurance team: https://infosec.mozilla.org/guidelines/web_security.html
3. Narzędzie Burp do testowania bezpieczeństwa aplikacji webowych: https://portswigger.net/burp
4. Historyczne przejęzyczenie: https://en.wikipedia.org/wiki/HTTP_referer