Bezpieczeństwo Aplikacji Internetowych

CSRF (Cross-Site Request Forgery)

CSRF (Cross-Site Request Forgery; alternatywnie używane nazwy: XSRF, session riding czy one-click attack) to cały czas dość częsta, a jednocześnie mało rozumiana podatność. Okazjonalnie bywa mylona z podatnością XSS, niekiedy jest prezentowana z innymi błędami bezpieczeństwa, co zaciemnia istotę problemu.

Czym jest CSRF? Definicja na stronie OWASP mówi mniej więcej tak:

> *jest to zmuszenie przeglądarki ofiary do wykonania pewnej nieautoryzowanej akcji (wykonania żądania HTTP), a atakujący na cel bierze zalogowanego użytkownika*[*].

Warto podkreślić, że jest to atak na przeglądarkę internetową ofiary, a nie na część serwerową aplikacji webowej; dla serwera żądania HTTP powstałe w wyniku ataku to zwykła komunikacja z przeglądarki użytkownika. Nieco bardziej zwięzłą definicję podaje serwis CWE (Common Weakness Enumeration):

> *[Aplikacja jest podatna na CSRF], kiedy nie sprawdza, czy wysłane do niej prawidłowe żądanie HTTP zostało świadomie wykonane przez użytkownika*[**].

Należy też uświadomić sobie, że CSRF jest podatnością wymagającą pewnego działania ofiary. Może nim być zwykłe korzystanie z aplikacji lub wejście na odpowiednio spreparowaną stronę WWW.

CSRF nie należy mylić z atakiem man-in-the-browser , w którym atakujący również może wpływać na działanie przeglądarki, ale wiąże się to z wcześniejszym zainstalowaniem w systemie ofiary malware'u. W przypadku CSRF system, jak i przeglądarka ofiary nie są w żaden sposób trwale modyfikowane. Wykorzystana jest tu po prostu pewna właściwość architektury web i przeglądarek internetowych.

CSRF to również nie to samo co XSS . Jeśli w aplikacji występuje XSS – to jest możliwość zrealizowania CSRF, ale jeśli nasza aplikacja podatna jest na CSRF, to niekoniecznie musimy być podatni na XSS. Dodatkowo sam Cross-Site Scripting może służyć do ominięcia metod ochrony przed CSRF. Poniżej zaprezentowane zostały przykłady realizacji takiego ataku.

PRZYKŁAD 1. CSRF REALIZOWANY W TEJ SAMEJ DOMENIE. NIEAUTORYZOWANE UTWORZENIE NOWEGO KONTA ADMINISTRACYJNEGO, METODA GET

W tym przypadku rozważmy aplikację (np. forum dyskusyjne), dostępną pod konkretną domeną (np. forum.training.securitum.com). Atakujący będzie chciał zmusić przeglądarkę administratora forum (wykorzystać podatność CSRF) do zarejestrowania nowego konta o uprawnieniach administracyjnych (z hasłem, które sam poda). Atak realizowany jest w kilku krokach:

1. Atakujący w komentarzu na forum umieszcza np. następujący tag:
< img
src="http://forum.training.securitum.com/admin/addUser?login=support&pass=1234&type
=adm">

2. Administrator uwierzytelnia się w aplikacji oraz wchodzi na stronę z moderacją komentarzy.

3. Do przeglądarki administratora ładuje się kod HTML z przesłanym wcześniej przez atakującego tagiem < img. Podczas próby pobrania obrazu wskazanego w tagu < img przeglądarka administratora realizuje automatycznie żądanie HTTP do panelu administracyjnego (jest to CSRF) – i tym samym tworzy nowe konto w systemie (konto ma uprawnienia administratora, natomiast atakujący zna hasło dostępowe).

Podsumowując:

1. W ataku nie został wykorzystany JavaScript.

2. W ataku nie została wykorzystana podatność XSS (gdyby występowała ona w aplikacji, można by również zrealizować CSRF, wykonując odpowiednie żądanie HTTP za pomocą JavaScript).

3. Atakujący nie znał loginu ani hasła administratora.

4. W logach serwera WWW jako źródłowy adres IP klienta, który dodał nieautoryzowanego użytkownika, widoczny będzie realny adres IP komputera atakowanego administratora (nie będzie to więc adres IP atakującego).

5. Atakujący nie widzi odpowiedzi na żądanie HTTP, do którego wykonania został zmuszony administrator, ale nie ma to wpływu na skuteczność ataku.

6. Atak odbywa się w ramach jednej domeny (forum.training.securitum.com) – czasem tego typu wariant CSRF nazywa się OSRF (On-site Request Forgery).

W takim scenariuszu może być atakowany w zasadzie dowolny panel administracyjny aplikacji webowej, który przetwarza (np. wyświetla) pewne dane przesłane przez

użytkownika. Może być to aplikacja zawierająca formularz kontaktowy (i wyświetlająca przesłane przez użytkowników sprawy w panelu webowym), aplikacja umożliwiająca wysyłanie podań o pracę itp.

W analizie tego przykładu widać, że aplikacje webowe domyślnie są podatne na CSRF (chyba że zostały wykorzystane stosowne biblioteki czy mechanizmy ochronne) – po prostu w ten sposób zaprojektowano architekturę web. Co się stanie w przypadku, gdy aplikacja będzie w odpowiedni sposób filtrowała wartości przekazywane przez użytkownika do formularza z komentarzem (tj. nie będzie możliwości wstrzyknięcia do przeglądarki administratora fragmentu HTML np. z tagiem < img/> Potencjalnie CSRF nadal jest możliwy – ale ścieżka ataku będzie wyglądać nieco inaczej.

PRZYKŁAD 2. CSRF REALIZOWANY POMIĘDZY RÓŻNYMI DOMENAMI. NIEAUTORYZOWANE USUNIĘCIE KONTA ADMINISTRATORA, METODA GET

Czasem wykorzystanie podatności CSRF przyjmuje bardzo prostą formę. Atakujący umieszcza w źródle HTML pewnej strony (np. atak.training.securitum.com) tag wyglądający w ten sposób:

```
<img src="https://forum.training.securitum.com/admin/delete_account">
```

i nakłania zalogowanego administratora do wejścia na swoją stronę.

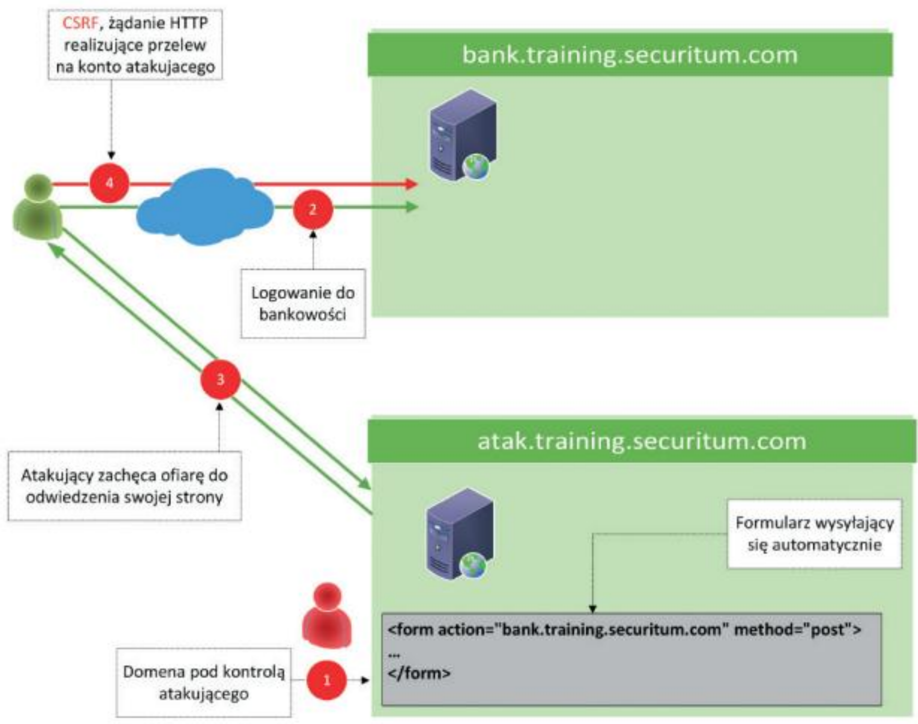
Zauważmy, że serwis atak.training.securitum.com może wyglądać całkiem normalnie, a w gąszczu różnych tagów ukryty jest ten jeden, który kasuje konto. Jeśli nie mamy żadnego zabezpieczenia przed CSRF ani potwierdzenia faktu usunięcia konta – może ono zostać automatycznie usunięte**.**

CSRF a Same-Origin Policy

W tym momencie można się zastanawiać, czy taka komunikacja nie łamie zasad określonych w Same-Origin Policy* ? Otóż nie. Mechanizm ten domyślnie pozwala na pewną komunikację pomiędzy różnymi originami. Na pewno wielu z nas widziało strony, które pobierają obrazy w tagu img z innej domeny i domyślnie nie ma z tym problemów. Inne w ten sposób zachowujące się tagi to np.: audio, video, frame, iframe, object.

PRZYKŁAD 3. CSRF REALIZOWANY POMIĘDZY RÓŻNYMI DOMENAMI. BANKOWOŚĆ ELEKTRONICZNA, METODA POST

Zobaczmy chyba najczęściej przytaczany przykład wykorzystania podatności CSRF – czyli z użyciem dwóch domen oraz formularza HTML wysyłającego dane metodą POST. Tym razem ofiarą będzie użytkownik bankowości elektronicznej:



Rysunek 1. Przykładowy atak CSRF na bankowość elektroniczną

Scenariusz ataku wygląda następująco:

1. Atakujący umieszcza w domenie atak.training.securitum.com formularz wysyłający dane do innej domeny metodą POST. OWASP podaje mniej więcej taki fragment HTML:

```
<body onload="document.formcsrf.submit();">
<form name="formcsrf" action="https://bank.training.securitum.com/ ⤦
transfer.do" method="POST">
<input type="hidden" name="dstacct" value="MULE"/>
<input type="hidden" name="srcacct" value="MARIA"/>
<input type="hidden" name="amount" value="100000"/>
<input type="submit" value="test"/>
</form>
</body>
```

2. Ofiara loguje się do bankowości elektronicznej.

3. Ofiara wchodzi w innej zakładce przeglądarki na atak.training.securitum.com (tutaj konieczny jest pewien rodzaj socjotechniki – służący nakłonieniu ofiary do wejścia w to miejsce).

4. Natychmiast po wejściu na stronę przeglądarka ofiary wysyła formularz wskazany w punkcie 1.

Oczywiście, zdecydowana większość systemów bankowości elektronicznej jest obecnie zarówno chroniona przed samą podatnością CSRF, jak i wymaga dodatkowej autoryzacji przy przelewie na nieznane konto – przynajmniej tyle mówi teoria.

Przykład ten rozwiewa często wspominany mit: miejsca w aplikacji, które przejmują dane tylko metodą POST, nie są podatne na CSRF. **Nie jest to prawda.**

Czasem aplikacja używa formularzy typu POST, ale jednocześnie te same wartości można wysłać metodą GET – z parametrami umieszczonymi w URL-u . Co takie zachowanie daje atakującemu? Nieco prostszy sposób przygotowania ataku – wystarczy, aby przeglądarka ofiary „zobaczyła" taki obrazek:

```
<img src="https://bank.training.securitum.com/transfer.do?srcacct= ⤦
MARIA&dstacct=MULE&amount=100000">
```

CSRF a inne niż GET/POST metody HTTP

Do tej pory widzieliśmy CSRF z wykorzystaniem metod GET oraz POST. Co z ewentualnymi innymi metodami? (np. PUT, DELETE). Formularze HTML nie pozwalają na używanie innych metod niż POST oraz GET, jednak istnieje pewna często wykorzystywana konwencja umożliwiająca ominięcie tego zabezpieczenia. Jest nią parametr _method. Jak wygląda jego zastosowanie?

```
<body onload="document.formcsrf.submit();">
<form name="formcsrf" action="https://bank.training.securitum.com/api/ ⤦
trusted_contrators/1" method="POST">
<input type="hidden" name="_method" value="DELETE"/>
<input type="submit" value="test"/>
</form>
</body>
```

```
<img src="https://bank.training.securitum.com/api/trusted_contrators/1?_method=DELETE">
```

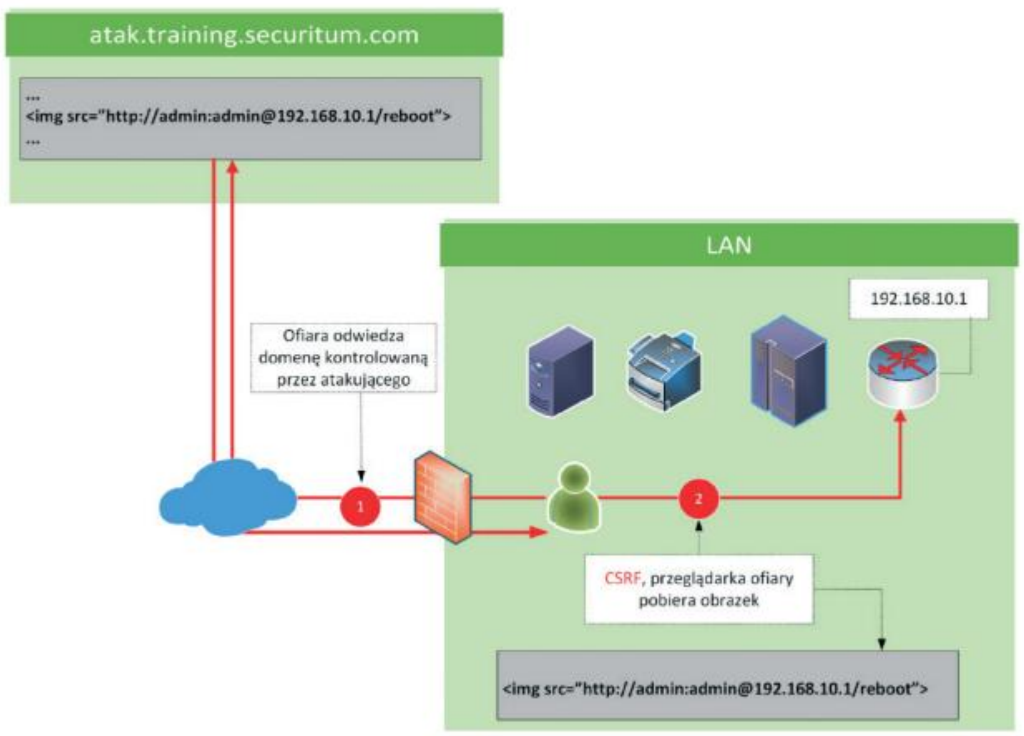
W tym przypadku jeśli API obsługuje metodę DELETE oraz wspiera „magiczny" parametr _method, to usunięty zostanie zaufany odbiorca o ID=1.

PRZYKŁAD 4. CSRF W POŁĄCZENIU Z INNYMI PODATNOŚCIAMI – URZĄDZENIA SIECIOWE

Pamiętajmy, że we wspomnianych w przykładzie 1 tagach HTML w parametrze src może znajdować się również adres z sieci prywatnej, np. :

```
<img src="http://admin:admin@192.168.10.1/reboot/">
```

W przykładzie tym widzimy również podanie użytkownika oraz hasła zgodnie ze schematem HTTP Basic Authentication.



Rysunek 2. Przykład ataku CSRF na urządzenie sieciowe

W tym miejscu zauważmy jeden ważny fakt – ofiara wcale nie musiała być zalogowana do panelu webowego urządzenia dostępnego pod adresem 192.168.10.1. Wystarczyło, że nie zostały tutaj zmienione domyślne dane dostępowe, a użytkownik, który wszedł na stronę ze „złośliwym" Tagiem img, znajdował się w stosownej sieci. Samo urządzenie, które zostanie

zrestartowane, wcale nie musi być dostępne bezpośrednio od strony Internetu, a jednak atakujący ma możliwość przesyłania do niego pewnych żądań HTTP.

PRZYKŁAD 5. PODATNOŚĆ WIELOETAPOWA – PRZEJĘCIE DOSTĘPU DO SYSTEMU WORDPRESS

W marcu 2019 roku opublikowano szczegóły podatności w WordPressie (wersje niższe niż 5.1.1) umożliwiającej przejęcie uprawnień administratora bez posiadania w początkowej fazie ataku żadnych danych dostępowych do systemu. Pierwszym elementem całego ciągu problemów była właśnie podatność CSRF:

> *Dodawanie nowego komentarza w WordPressie nie jest zabezpieczone przed CSRF. Przyczyną tego stanu są funkcje typu trackback czy pingback, które mogłyby działać nieprawidłowo z takim zabezpieczeniem. Oznacza to, że atakujący może dodać nowy komentarz – np. jako administrator WordPressa – korzystając z CSRF[**].*

Zatem atakujący mógł stworzyć stosowną stronę z formularzem HTML typu POST oraz w pewien sposób zachęcić do jej odwiedzenia administratora WordPressa. W trakcie tych odwiedzin formularz HTML był automatycznie wysyłany, co skutkowało dodaniem komentarza przez administratora-ofiarę. Raczej nic groźnego, prawda? Jednak kolejna podatność umożliwiała wstrzyknięcie fragmentu kodu HTML do komentarza (dało się to zrobić tylko w przypadku, gdy komentarz stworzył administrator – nie zwykły użytkownik). Można to było zrealizować np. w taki sposób: .
< a title='XSS " onmouseover=evilCode() id=" '>.

Na pierwszy rzut oka ponownie nie jest to nic groźnego, bo taki fragment HTML nie powoduje uruchomienia funkcji evilCode() z poziomu JavaScript. Jednak przed zapisaniem komentarza do bazy WordPress przygotowywał link w taki sposób, aby był przyjazny do celów SEO (Search Engine Optimization). W szczególności każdy parametr tagu był jeszcze raz otaczany podwójnymi cudzysłowami. Ostatecznie wyglądało to w ten sposób: .
< a title="XSS " onmouseover=evilCode() id=" ">.
Czyli mamy już podatność XSS, z której wykorzystaniem możliwe było np. automatyczne dodanie nowego „pluginu” do WordPressa, będącego w przypadku ataku backdoorem dającym dostęp na poziomie systemu operacyjnego.

Zauważmy, że obecny przykład to cała seria błędów (w tym podatność CSRF) prowadząca do możliwości pełnego przejęcia serwisu opartego na WordPressie. CSRF był tutaj tylko pierwszym (ale bardzo istotnym) krokiem.

Analizując opisy podatności CSRF, warto się zastanowić, czy mówimy tylko o jednej podatności czy o całej serii oraz w którym miejscu realnie wykorzystywany jest Cross-Site Request Forgery.

METODY OCHRONY PRZED CSRF

Ochronę możemy zrealizować na kilka alternatywnych sposobów , przy czym warto pamiętać, że najistotniejsze jest zabezpieczenie miejsc aplikacji realizujących zdarzenia modyfikujące pewne wartości w systemie (np. tworzące użytkownika, zmieniające hasło itp.). Atakujący, z jednej strony, nie widzi odpowiedzi na żądanie HTTP, do którego została zmuszona ofiara, więc podatność CSRF występująca np. w funkcji listowania informacji o przelewach niewiele daje.
Z drugiej strony – konsekwentne wdrożenie ochrony w całej aplikacji ułatwia uniknięcie niebezpiecznych wyjątków oraz objęcie ochroną również nowych funkcji w systemie.

Losowe tokeny

Pierwszą zalecaną przez OWASP metodą jest Synchronizer Token Pattern, czyli użycie losowych tokenów (ciągów znaków) związanych z zalogowaną sesją użytkownika.

Podczas sesji użytkownika generowany jest odpowiednio długi, pseudolosowy ciąg znaków, który przekazywany jest do kolejnych żądań, np. w taki sposób:

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken" value="OWY4NmQwODE4ODRjN2Q2NTlhM ↲
mZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZMGYwMGEwOA==">
[...]
</form>
```

Strona obsługująca formularz musi z kolei sprawdzić, czy przekazany token to rzeczywiście wartość, która została wygenerowana przez aplikację, i czy jest powiązana z danym użytkownikiem.

Atakujący, z jednej strony, nie zna oczywiście tokena wygenerowanego dla konkretnego użytkownika, więc nie jest w stanie przygotować działającego formularza wykorzystującego CSRF. Z drugiej strony, zauważmy, że wyciek tokena powoduje możliwość ominięcia ochrony przed CSRF.

W tym miejscu jako osobny wątek warto rozważyć żądania HTTP typu GET (te, które nie są zwykłym, nic niezmieniającym w aplikacji odczytem). Z jednej strony, można dać zalecenie:

dodawajmy do takich żądań token. Z drugiej strony, w przypadku metody GET trzeba pamiętać o możliwym wycieku tokena. Parametry URL są widoczne w pasku przeglądarki, zapisywane w historii przeglądarki/logach webserwera czy przekazywane w nagłówku Referer do serwisu, który jest linkowany w naszej aplikacji.

Obecnie OWASP zaleca zmianę wszystkich żądań metodą GET (które zmieniają stan aplikacji) na POST. Przy okazji należy sprawdzić, czy nie można sztucznie zmienić w aplikacji żądania POST na żądanie GET (z parametrami przekazywanymi w URI)

Innym scenariuszem, w którym atakujący może uzyskać dostęp do tokena (i ominąć ochronę przeciwko CSRF), jest wykorzystanie podatności XSS. W tym przypadku napastnik realizuje dwa kroki:

1. Pobranie tokena z wykorzystaniem XSS (JavaScript).

2. Wygenerowanie żądania HTTP (np. za pomocą tego samego XSS) z osadzonym już tokenem.

Zatem jeśli mamy w naszej aplikacji XSS, najprawdopodobniej będzie się dało ominąć ochronę przed CSRF.

W rekomendacjach OWASP możemy znaleźć dodatkowe warianty budowania tokenów (z wykorzystaniem szyfrowania czy algorytmu HMAC). Zaznaczmy, że czasem frameworki dostarczają możliwości automatycznej ochrony przed CSRF. Przy czym warto mieć świadomość, przed czym jesteśmy chronieni, a przed czym nie. Przykładowo Microsoft informuje:

> *ASP.NET nie wspiera automatycznego dodawania tokenów anty-CSRF do żądań typu `GET`**.*

Z kolei w Django mamy rozsądną informację (a zarazem ostrzeżenie, jak nie używać wbudowanego mechanizmu ochrony przeciwko CSRF):

> *W przypadku szablonów, które używają formularzy typu `POST`, użyj tagu `csrf_token` w środku elementu `<form>`, jeśli kieruje on do wewnętrznego URL-a, np.: `<form method="post">{% csrf_token %}`. Ta technika nie powinna być jednak wykorzystywana w formularzach, które kierują do zewnętrznych URL-i – token będzie wtedy wyciekał*.*

Tokeny anty-CSRF miały być traktowane jako poufne, więc nie można ich wysyłać do innych serwisów.

Warto również zwrócić uwagę na to, co oznacza pojęcie automatyczna ochrona przeciwko CSRF dostępna we frameworku. Czy jest ona automatycznie włączona? A może

automatyczne jest tylko generowanie (i sprawdzanie) tokenów – jeśli programista świadomie użyje odpowiedniej funkcji frameworka?

Samo generowanie tokenów anty-CSRF nie rozwiązuje problemu – należy również koniecznie sprawdzać ich poprawność. Niektórzy twórcy CMS-ów na zgłoszenie podatności CSRF reagują np. tak:

> *tokeny anty-CSRF mamy, ale jakoś tak wyszło, że zapomnieliśmy sprawdzać ich poprawność po stronie serwerowej**.*

SameSite

Ciekawą, nową formą ochrony przed CSRF jest atrybut SameSite dodawany do ciasteczek. Odpowiednia konfiguracja blokuje wysyłanie ciastka sesyjnego, jeśli zapytanie realizowane jest pomiędzy domenami (czyli w naszym przypadku najczęściej wykorzystywany wariant – formularz typu POST znajdujący się w domenie kontrolowanej przez atakującego). W tym przypadku ofiara nadal wysyła złośliwie przygotowane żądanie HTTP do aplikacji, ale nie jest ono uwierzytelnione (czyli nie ma możliwości wprowadzania zmian w aplikacji).

Mechanizm wydaje się rozsądny w kontekście ochrony przed CSRF, ale nie zaleca się go obecnie jako jedynej metody zabezpieczenia aplikacji przed tą podatnością. Zauważmy też, że SameSite nie ochroni nas przed scenariuszem ataku opisanym w przykładzie 1 (bo zapytanie wysyłane jest w ramach tej samej domeny) oraz 4 (bo atak nie opiera się na ciasteczkach).

Ekran logowania

OWASP, definiując podatność CSRF, wskazuje, że atak odbywa się na zalogowanego użytkownika.

Przywoływana jest tutaj konkretna, historyczna podatność w Google , która realizowana była według scenariusza:

1. Atakujący zakłada swoje konto w Google.

2. Atakujący przygotowuje automatycznie wysyłający się formularz typu POST, zawierający własne dane uwierzytelniające i w pewien sposób namawia ofiarę do wejścia na stronę z tym formularzem (klasyczny element podczas wykorzystania podatności CSRF).

3. Ofiara jest automatycznie logowana, wyszukuje pewne treści.

4. Atakujący posiada dostęp do historii wyszukiwania ofiary (znajduje się ona bezpośrednio na koncie Google atakującego).

Nowe podatności wprowadzone przez ochronę przeciwko CSRF

Warto mieć świadomość, że każdy nowy element aplikacji czy parametr – to potencjalnie nowe problemy bezpieczeństwa. Nie inaczej jest z tokenami CSRF, choć nowe podatności pojawiają się tutaj niezmiernie rzadko.

Przykładem niech będzie podatność zgłoszona do firmy Uber . W tym przypadku parametr state w implementacji OAuth 2.0, w którym normalnie powinien się znaleźć token anty-CRSF, zawierał adres URL, do którego przekierowywany był użytkownik. Umożliwiło to wykradanie tokenów OAuth 2.0.

LITERATURA

Bezpieczeństwo Aplikacji Webowych - Michał Bentkowski / Artur Czyż / Rafał 'bl4de' Janicki / Jarosław Kamiński Adrian 'vizzdoom' Michalczyk / Mateusz Niezabitowski / Marcin Piosek Michał Sajdak / Grzegorz Trawiński / Bohdan Widła - ISBN: 978-83-954853-2-9 - Kraków 2020