

(Nie)bezpieczeństwa na froncie

Kurs Tworzenia Aplikacji Frontendowych 2022



Uniwersytet
Wrocławski

Popularne ataki



Uniwersytet
Wrocławski

Popularne ataki

- **Clickjacking**, gdzie atakujący "przebiera" aplikację tak, by ofiara kliknęła w to, w co atakujący chce.



Popularne ataki

- **Clickjacking**, gdzie atakujący "przebiera" aplikację tak, by ofiara kliknęła w to, w co atakujący chce.
- **Cross-Site Request Forgery (CSRF)**, gdzie atakujący wykonuje niepożądane ale autoryzowane zapytania (np. poprzez ciasteczka) w imieniu użytkownika.



Popularne ataki

- **Clickjacking**, gdzie atakujący "przebiera" aplikację tak, by ofiara kliknęła w to, w co atakujący chce.
- **Cross-Site Request Forgery (CSRF)**, gdzie atakujący wykonuje niepożądane ale autoryzowane zapytania (np. poprzez ciasteczka) w imieniu użytkownika.
- **Cross-Site Scripting (XSS)**, gdzie atakujący umieszczają pewną treść w aplikacji tak, by przeglądarki innych użytkowników (ofiar) uruchamiając ją, wykonały jakiś kod.



Popularne ataki

- **Clickjacking**, gdzie atakujący "przebiera" aplikację tak, by ofiara kliknęła w to, w co atakujący chce.
- **Cross-Site Request Forgery (CSRF)**, gdzie atakujący wykonuje niepożądane ale autoryzowane zapytania (np. poprzez ciasteczka) w imieniu użytkownika.
- **Cross-Site Scripting (XSS)**, gdzie atakujący umieszczają pewną treść w aplikacji tak, by przeglądarki innych użytkowników (ofiar) uruchamiając ją, wykonały jakiś kod.
- **Denial of Service (DoS)**, gdzie atakujący zaburza dostępność jakiegoś zasobu (np. serwera).



Clickjacking



Uniwersytet
Wrocławski

Clickjacking

Atakujący na swojej stronie osadza jakąś *interesującą* treść (np. "Darmowy iPhone 14!!!"), przez którą przechodzą kliknięcia (`pointer-events: none`) i trafiają w `<iframe>` z atakowaną aplikacją tam, gdzie atakujący chce (np. "Polub nasz szemrany post").



Clickjacking

Atakujący na swojej stronie osadza jakąś *interesującą* treść (np. "Darmowy iPhone 14!!!"), przez którą przechodzą kliknięcia (`pointer-events: none`) i trafiają w `<iframe>` z atakowaną aplikacją tam, gdzie atakujący chce (np. "Polub nasz szemrany post").

Można się bronić ustawiając konkretne nagłówki (`Content-Security-Policy`) lub obchodząc to specjalnym kodem który sprawdzi, czy aplikacja jest osadzona (i np. zablokuje wtedy pewne operacje).



Cross-Site Scripting (XSS)



Uniwersytet
Wrocławski

Cross-Site Scripting (XSS)

Skoro wykonujemy kod w czyjejś przeglądarce, to mamy dostęp do wszystkiego co ta nam oferuje, czyli możemy...



Uniwersytet
Wrocławski

Cross-Site Scripting (XSS)

Skoro wykonujemy kod w czyjejś przeglądarce, to mamy dostęp do wszystkiego co ta nam oferuje, czyli możemy...

- Wykonywać zapytania HTTP ([Fetch API](#)).



Cross-Site Scripting (XSS)

Skoro wykonujemy kod w czyjejś przeglądarce, to mamy dostęp do wszystkiego co ta nam oferuje, czyli możemy...

- Wykonywać zapytania HTTP ([Fetch API](#)).
- Nasłuchiwać na wszelkie zdarzenia, implementując pełnoprawny keylogger.



Cross-Site Scripting (XSS)

Skoro wykonujemy kod w czyjejś przeglądarce, to mamy dostęp do wszystkiego co ta nam oferuje, czyli możemy...

- Wykonywać zapytania HTTP ([Fetch API](#)).
- Nasłuchiwać na wszelkie zdarzenia, implementując pełnoprawny keylogger.
- Czytać i modyfikować wyświetlany HTML ale też `document.cookie` czy `localStorage`.



Cross-Site Scripting (XSS)

Skoro wykonujemy kod w czyjejś przeglądarce, to mamy dostęp do wszystkiego co ta nam oferuje, czyli możemy...

- Wykonywać zapytania HTTP ([Fetch API](#)).
- Nasłuchiwać na wszelkie zdarzenia, implementując pełnoprawny keylogger.
- Czytać i modyfikować wyświetlany HTML ale też `document.cookie` czy `localStorage`.
- Skorzystać z mniej znanych API: grzebać w systemie plików ([File and Directory Entries API](#)), komunikować się z urządzeniami po Bluetooth ([Web Bluetooth API](#)), wysłać sygnał NFC ([Web NFC API](#)), syntezyzować mowę ([Web Speech API](#)), wibrować ([Vibration API](#))...



Cross-Site Scripting (XSS)



Uniwersytet
Wrocławski

Cross-Site Scripting (XSS)

Rozróżniamy dwa (główne) rodzaje, zależne od tego, skąd bierze się wstrzyknięty kod.



Cross-Site Scripting (XSS)

Rozróżniamy dwa (główne) rodzaje, zależne od tego, skąd bierze się wstrzyknięty kod.

Persistent (lub **stored**) **XSS**,
gdzie złośliwy kod znajduje
się w jakiś trwałym miejscu
(np. serwerze).



Cross-Site Scripting (XSS)

Rozróżniamy dwa (główne) rodzaje, zależne od tego, skąd bierze się wstrzyknięty kod.

Persistent (lub **stored**) **XSS**,
gdzie złośliwy kod znajduje
się w jakiś trwałym miejscu
(np. serwerze).

Reflected XSS, gdzie
złośliwy kod znajduje się w
innym, nietrwałym miejscu
(np. linku).



Persistent XSS



Uniwersytet
Wrocławski

Persistent XSS

```
async function loadCommentText(  
  list: HTMLUListElement,  
  id: string,  
) {
```



Persistent XSS

```
async function loadCommentText(  
  list: HTMLUListElement,  
  id: string,  
) {  
  // Load comment from the API.  
  const url = `https://example.com/comments/${id}`;  
  const response = await fetch(url);  
  const { html } = await response.json();
```



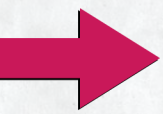
Persistent XSS

```
async function loadCommentText(  
  list: HTMLUListElement,  
  id: string,  
) {  
  // Load comment from the API.  
  const url = `https://example.com/comments/${id}`;  
  const response = await fetch(url);  
  const { html } = await response.json();  
  
  // Create list item and add it to the list.  
  const node = document.createElement('li');  
  node.innerHTML = html;  
  list.appendChild(commentNode);  
}
```



Persistent XSS

```
async function loadCommentText(  
  list: HTMLUListElement,  
  id: string,  
) {  
  // Load comment from the API.  
  const url = `https://example.com/comments/${id}`;  
  const response = await fetch(url);  
  const { html } = await response.json();  
  
  // Create list item and add it to the list.  
  const node = document.createElement('li');  
  node.innerHTML = html;  
  list.appendChild(commentNode);  
}
```



Reflected XSS



Uniwersytet
Wrocławski

Reflected XSS

```
async function renderSearchHeader() {  
  // Get the search term from  
  const url = new URL(window.location);  
  const searchTerm = url.searchParams.get('search');
```



Reflected XSS

```
async function renderSearchHeader() {  
  // Get the search term from  
  const url = new URL(window.location);  
  const searchTerm = url.searchParams.get('search');  
  
  // Find the existing header.  
  const selector = '#search-header';  
  const header = document.querySelector(selector);
```

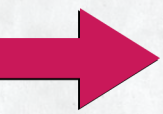


Reflected XSS

```
async function renderSearchHeader() {  
  // Get the search term from  
  const url = new URL(window.location);  
  const searchTerm = url.searchParams.get('search');  
  
  // Find the existing header.  
  const selector = '#search-header';  
  const header = document.querySelector(selector);  
  
  // Render it.  
  const html = `Results for <b>${searchTerm}</b>`;   
  header.innerHTML = html;  
}
```



Reflected XSS

```
async function renderSearchHeader() {  
  // Get the search term from  
  const url = new URL(window.location);  
  const searchTerm = url.searchParams.get('search');  
  
  // Find the existing header.  
  const selector = '#search-header';  
  const header = document.querySelector(selector);  
  
  // Render it.  
  const html = `Results for <b>${searchTerm}</b>`;   
  header.innerHTML = html;  
}
```



Nie tylko innerHTML



Uniwersytet
Wrocławski

Nie tylko innerHTML

Zdaje się najczęstszym źródłem tej podatności jest brak sanitizacji ("czyszczenia") kodu HTML, który pochodzi od użytkownika. Ale są też inne sposoby:



Nie tylko innerHTML

Zdaje się najczęstszym źródłem tej podatności jest brak sanitizacji ("czyszczenia") kodu HTML, który pochodzi od użytkownika. Ale są też inne sposoby:

Różne atrybuty DOM, np.
href w tagu <a> czy src i
onerror w tagu .



Nie tylko innerHTML

Zdaje się najczęstszym źródłem tej podatności jest brak sanitizacji ("czyszczenia") kodu HTML, który pochodzi od użytkownika. Ale są też inne sposoby:

Różne atrybuty DOM, np. href w tagu <a> czy src i onerror w tagu .

```
<a href="javascript:alert('Boom!')">  
    Login  
</a>  

```



Nie tylko innerHTML

Zdaje się najczęstszym źródłem tej podatności jest brak sanitizacji ("czyszczenia") kodu HTML, który pochodzi od użytkownika. Ale są też inne sposoby:

Różne atrybuty DOM, np. href w tagu <a> czy src i onerror w tagu .

```
<a href="javascript:alert('Boom!')">  
  Login  
</a>  

```

eval, setInterval i
setTimeout.



Nie tylko innerHTML

Zdaje się najczęstszym źródłem tej podatności jest brak sanitizacji ("czyszczenia") kodu HTML, który pochodzi od użytkownika. Ale są też inne sposoby:

Różne atrybuty DOM, np. href w tagu <a> czy src i onerror w tagu .

```
<a href="javascript:alert('Boom!')">  
  Login  
</a>  

```

eval, setInterval i
setTimeout.

```
setTimeout("alert('Boom!')", 1000);
```



A nawet CSS



Uniwersytet
Wrocławski

A nawet CSS

```
function defineUserColor(color: string) {  
  const style = document.createElement('style');  
  style.textContent = `  
    .user-color {  
      color: ${color};  
    }  
  `;  
}
```



A nawet CSS

```
function defineUserColor(color: string) {  
  const style = document.createElement('style');  
  style.textContent = `  
    .user-color {  
      color: ${color};  
    }  
  `;  
}  
  
defineUserColor(`red`;  
  background: url("https://ip.tracking.evil.com")  
`);
```



Jak się bronić?



Uniwersytet
Wrocławski

Jak się bronić?

Wszystko zależy od tego, jaką konkretnie podatność mamy. Podstawą natomiast powinien być **bezwzględny** zakaz ufania danym pochodzącym od użytkownika.



Uniwersytet
Wrocławski

Jak się bronić?

Wszystko zależy od tego, jaką konkretnie podatność mamy. Podstawą natomiast powinien być **bezwzględny** zakaz ufania danym pochodzącym od użytkownika.

Przykładowo, jeżeli wyświetlamy jakąś treść, użyjmy `textContent` zamiast `innerHTML`. Jeżeli faktycznie potrzebujemy tam HTML (np. edytor komentarzy pozwala formatować tekst), to musimy go odpowiednio zabezpieczyć.



Sanityzacja HTML



Uniwersytet
Wrocławski

Sanityzacja HTML

```
// There are tens of packages that sanitize the HTML.  
// This is just an example!  
import sanitizeHTML from 'sanitize-html';  
  
const safeHTML = sanitizeHTML(unsafeHTML, {  
  // Other tags are stripped.  
  allowedTags: ['a', 'b', 'i'],  
  // Other attributes are stripped.  
  allowedAttributes: { a: ['href'] },  
  // The `href` tag has to start with `https://`.  
  // Other URL schemes are stripped entirely.  
  allowedSchemes: ['https'],  
});
```



Denial of Service (DoS)



Uniwersytet
Wrocławski

Denial of Service (DoS)

Jak można się domyśleć, najczęstszym celem tego ataku są serwery i atakujący jest dla nas stroną trzecią. Może się natomiast zdarzyć tak, że to my sami wykonamy taki "atak" na naszej aplikacji *przypadkiem*.



Denial of Service (DoS)

Jak można się domyśleć, najczęstszym celem tego ataku są serwery i atakujący jest dla nas stroną trzecią. Może się natomiast zdarzyć tak, że to my sami wykonamy taki "atak" na naszej aplikacji *przypadkiem*.

Jako przykład weźmy listę 8 (Pokédex). Wyobraźmy sobie, że chcielibyśmy wyświetlić obrazki nie tylko po kliknięciu ale już na samej liście.



Denial of Service (DoS)

Jak można się domyśleć, najczęstszym celem tego ataku są serwery i atakujący jest dla nas stroną trzecią. Może się natomiast zdarzyć tak, że to my sami wykonamy taki "atak" na naszej aplikacji *przypadkiem*.

Jako przykład weźmy listę 8 (Pokédex). Wyobraźmy sobie, że chcielibyśmy wyświetlić obrazki nie tylko po kliknięciu ale już na samej liście.

Wymaga to od nas $N+1$ zapytań, gdzie N to liczba Pokémonów. Pierwsze, pobierające listę, musi się zakończyć najpierw, natomiast pozostałe mogą być wykonane równolegle.



Przykład



Uniwersytet
Wrocławski

Przykład

```
function renderPokemonList(limit: number) {  
  const api = 'https://pokeapi.co/api/v2/pokemon';  
  const url = `${api}?limit=${limit}`;  
  const response = await fetch(url);  
  const list = await response.json();  
}
```



Przykład

```
function renderPokemonList(limit: number) {  
  const api = 'https://pokeapi.co/api/v2/pokemon';  
  const url = `${api}?limit=${limit}`;  
  const response = await fetch(url);  
  const list = await response.json();  
  
  const listWithSprites = Promise.all(  
    list.map(async ({ name, url }) => {  
      const response = await fetch(url);  
      const { sprites } = await response.json();  
      return { sprites, name };  
    })  
  );  
};
```



Przykład

```
function renderPokemonList(limit: number) {  
  const api = 'https://pokeapi.co/api/v2/pokemon';  
  const url = `${api}?limit=${limit}`;  
  const response = await fetch(url);  
  const list = await response.json();  
  
  const listWithSprites = Promise.all(  
    list.map(async ({ name, url }) => {  
      const response = await fetch(url);  
      const { sprites } = await response.json();  
      return { sprites, name };  
    })  
  );  
  
  listWithSprites.forEach(renderPokemon);  
}
```



Przykład

```
function renderPokemonList(limit: number) {  
  const api = 'https://pokeapi.co/api/v2/pokemon';  
  const url = `${api}?limit=${limit}`;  
  const response = await fetch(url);  
  const list = await response.json();  
  
  const listWithSprites = Promise.all(  
    list.map(async ({ name, url }) => {  
      const response = await fetch(url);  
      const { sprites } = await response.json();  
      return { sprites, name };  
    })  
  );  
  
  listWithSprites.forEach(renderPokemon);  
}
```



Co z tym zrobić?



Uniwersytet
Wrocławski

Co z tym zrobić?

Przed wszystkim nasz serwer powinien sobie *jakoś* poradzić. Niezależnie od tego, czy to przez przypadek czy specjalnie, natłok równoległych zapytań nie powinien powodować problemów.



Co z tym zrobić?

Przede wszystkim nasz serwer powinien sobie *jakoś* poradzić. Niezależnie od tego, czy to przez przypadek czy specjalnie, natłok równoległych zapytań nie powinien powodować problemów.

Jeżeli chodzi o sam frontend, to sama przeglądarka już *trochę* pomaga, bo ogranicza ilość równoległych zapytań. Możemy to kontrolować sami, kolejując je odpowiednio.



Co z tym zrobić?

Przed wszystkim nasz serwer powinien sobie *jakoś* poradzić. Niezależnie od tego, czy to przez przypadek czy specjalnie, natłok równoległych zapytań nie powinien powodować problemów.

Jeżeli chodzi o sam frontend, to sama przeglądarka już *trochę* pomaga, bo ogranicza ilość równoległych zapytań. Możemy to kontrolować sami, kolejując je odpowiednio.

Bardzo prostym i skutecznym rozwiązaniem jest paczka `p-limit`, która ogranicza ilość równoległe uruchomionych Promise'ów.



OWASP



Uniwersytet
Wrocławski

OWASP

Open Web Application Security Project to fundacja o jasnym celu: poprawa bezpieczeństwa oprogramowania.



Uniwersytet
Wrocławski

OWASP

Open Web Application Security Project to fundacja o jasnym celu: poprawa bezpieczeństwa oprogramowania.

To co jest dla nas najbardziej interesujące, to obszerne podsumowania, jak na przykład to: [Cross Site Scripting Prevention Cheat Sheet](#).



OWASP

Open Web Application Security Project to fundacja o jasnym celu: poprawa bezpieczeństwa oprogramowania.

To co jest dla nas najbardziej interesujące, to obszerne podsumowania, jak na przykład to: [Cross Site Scripting Prevention Cheat Sheet](#).

Ich lista [OWASP Top Ten](#) jest praktycznie standardem jeżeli chodzi o audyty bezpieczeństwa i choć mało z nich dotyczy frontendu, to często jest on pomijany a w rezultacie bardzo podatny.



Fin



Uniwersytet
Wrocławski