

Session'ı ve CSRF Zafiyetini Anlamak & SameSite Cookie Önlemi | MDISEC Neler Anlattı #4



İLKER YILMAZ

8 min read · Just now

Herkese Merhaba, serimizin sıradaki yazısına hoşgeldiniz. Bu ders kapsamında da Session ve CSRF hakkında öğrendiğimiz bilgilerin notunu bu yazıda bulabilirsiniz. Serimizin devamı gelecektir, beklemede kalınız.

Önceki serilerde de yaptığımız gibi öncelikle meselenin altyapısını öğrenerek ilerleyelim. HTTP hakkında konuşalım öncelikle.

HTTP Hakkında

HTTP'nin temelde ortaya çıkış hikayesi basit ihtiyaçların giderilmesine yönelik olduğu için çok etkili bir protokol değildir. Her zaman için bir request karşılığında bir response alınır. Yani her şey request-response arasında yaşanmaktadır.

Farklı OSI katmanlarındaki protokoller birbiriyle kıyaslanmasa da şöyle bir kıyaslama yapalım.

HTTP dediğimizde aklımıza TCP 3-Way Handshake gelebilir.

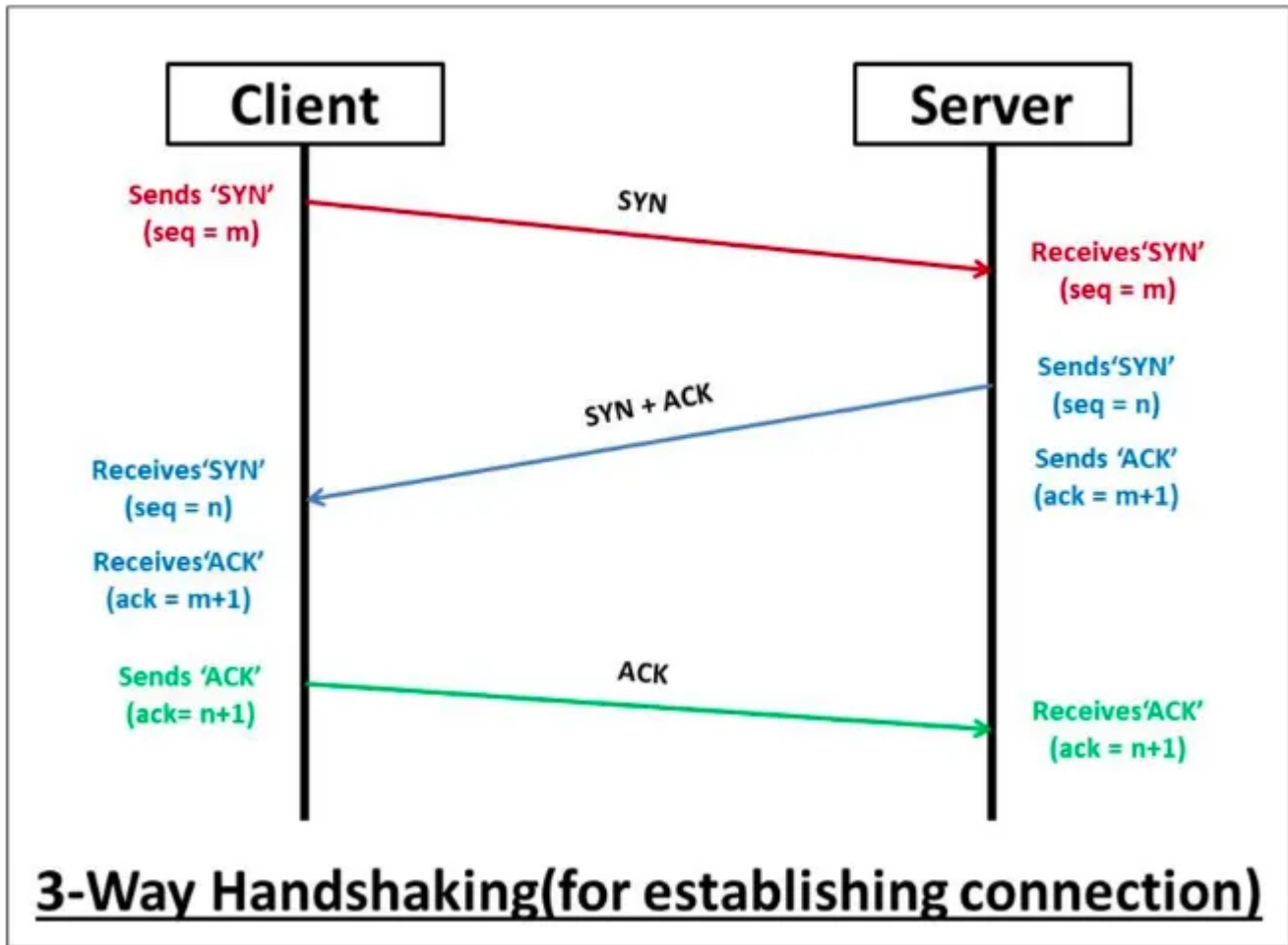


image source: <https://afteracademy.com/blog/what-is-a-tcp-3-way-handshake-process/>

Burada TCP 3-Way Handshake yapısını görebilirsiniz. Peki böyle bir yapıya neden ihtiyaç var?

Öncelikle bu yapının en büyük katkısı işlemler tamamlandığı zaman karşımızda konuşmak istediğimiz kişiyi doğrulamış olmamızdır.

TCP 3-Way Handshake Nedir?

Bu yapıda bir canlandırma yapacak olursak şu şekilde bir konuşma geçmektedir;

Client: “Merhaba, seninle konuşmak istiyorum.”

Server: “Merhaba benimle konuşmak istediğini duydum ve seninle konuşmaya müsaitim”

Client: “Merhaba, ben seninle konuşmak istediğimi söylemiştim sen de bunu duymuşsun ve benimle konuşmak için müsait olduğunu duydum. Hadi konuşalım”

Bu şekilde 3'lü paket gönderilmektedir. En büyük katkısı karşı taraftaki insanı doğrulamasıdır. (İnsan diyerek benzetme yapılmıştır.)

Bunu HTTP ile kıyasladığımızda HTTP'deki en büyük eksikliklerden biri protokolda authentication olmamasıdır. Bir diğer konu da state'lerdir. (Not: farklı katmanlardaki protokolleri kıyaslamak her ne kadar doğru olmasa da bir özellik için örnek vermekteyiz.)

Burada görmüş olduğunuz request bir HTTP Post request'idir.

```
POST /index.html HTTP/1.1
Host: mdisec.com
User-Agent:

{'name': 'mehmet'}
```

Bu post request'i Header ve Body olmak üzere 2 kısımdan oluşmaktadır.

Peki Neden Header ve Body Kısımları Var ?

Bir örnek verecek olursak insan olarak bir baş ve bir vücuda sahibiz. Kafa kısmında önemli bilgiler bulunmaktadır. Bunun haricindeki bilgiler de body yani vücut tarafına gönderilir. Önemli bilgiler her zaman header'da gönderilir bundan dolayı. Örneğin cookie'nin neden header'da gittiğini artık daha iyi anlayabiliriz.

Genellikle 4 terim duyarız;

HTTP Request'inin Body ve Header'ı,

Bu request'e karşılık olarak gelen Response'un Body ve Header'ı.

HTTP ve HTTPS arasındaki fark nedir?

HTTPS'deki tek fark gelip giden verilerin şifrelenmesidir.

HTTP mekanizması içerisinde bir authentication desteği yoktur, peki günümüzde authentication mekanizması nasıl ilerlemektedir?

Bu kısımda da internet tarayıcıları ile HTTP protokolünün kendi arasında anlaştığı bir yapıyı canlandırılım zihnimizde.

Bu yapıda HTTP'nin header'ında özel bir alan bulunmaktadır. Cookie alanı. Bu Cookie'ler sunucu tarafında oluşturulduktan sonra browser'ın yerel veritabanında bulunsun diye bir kabul vardır.

```
1. Request
POST /login HTTP/1.1
Host: mdisec.com

username=mehmet&password=twitch
```

Buradaki HTTP Request'ine bir cevap yani Response geldiğini düşünelim.

```
HTTP 302 OK
Location: mdisec.com/dashboard
Set-Cookie: SESSION=afdjaskgpajgkfşagjkfalşgj
```

Günümüzde HTTP içerisinde bir authentication mekanizması olmadığı için bazı işlemler yapmamız gerekmektedir.

Burada request'i gönderen ve alan kişi browser'dır. Browser response'un header'ında Set-Cookie diye bir değer gördüğünde buradaki bilgiyi browser'ın yerel veritabanına kaydeder. Burada bu cookie'yi kaydetmek zorundadır çünkü 2. request isteği atılırken Cookie alanı otomatik olarak browser tarafından eklenir. Bu sayede siz her HTTP talebinde kullanıcı adı ve parola vermekten kurtulmuş olursunuz. Aksi halde her HTTP request'inde kullanıcı adı ve parola koyarak iletmeniz gerekirdi. Çünkü HTTP protokolü sizinle ilgili bilgileri unutmaktadır. Request-Response döngüsü ilk iletim için yaşanıp bitmektedir. Bir sonraki request'te bir önceki request'in ne olduğuna dair hiçbir bilgi bulunmamaktadır. Ancak Application katmanının sizi tanımaya devam etmesi gerekmektedir. Çünkü giriş yaptıktan sonra sizin kim olduğunuzu bilmesi gerekmektedir. Bunu sağlamak için de browser'larda protokolün de yapısında bulunan böyle bir düzen bulunmaktadır.

Böyle bir yapıda aklımıza yeni bir soru gelmektedir.

2. Request

GET /dashboard HTTP/1.1

Host:mdisec

Cookie: afdjaskgpajgkfşagjkfalşgj {keyword: HTTP Security Headers}

Browser'ın bir sonraki request'e bahsettiğimiz bilgileri eklemesi gerektiğini söylemiştik. Peki bu neye göre gerçekleşir? Giden tüm HTTP Request'lerine mi ekleme yapar yoksa bu işlem bir kurala göre mi yapılmaktadır ?

http://www.mdisec.com:80/

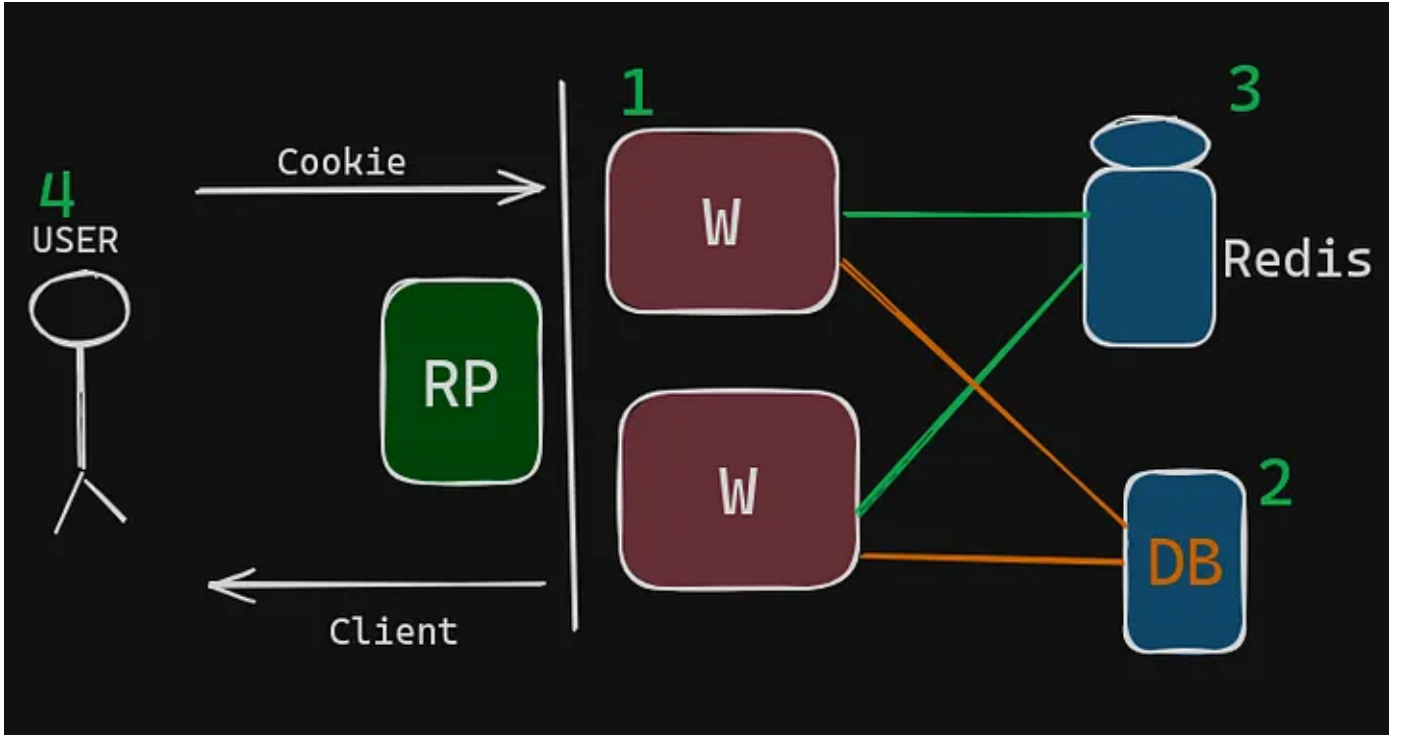
Browser'ların baktığı kısım burada “protocol”+”domain”+”port” kısımlarıdır. Bir web HTTP Request'i gittikten sonra HTTPS ile girdiğimizde oturum kayıplarıyla karşılaşmış olabilirsiniz. Sebebi bu durumdan kaynaklanmaktadır.

Artık bu kısma kadar şöyle bir çıkarım yapabiliriz;

Biz kullanıcı adı ve parolamızı verdikten sonra uygulama sunucusu bunun karşılığında bize bir kimlik vermektedir. ID gibi düşünebiliriz bunu. Burada bir takas söz konusudur. Verdiğimiz bilgilere karşılık olarak yapılan kontroller neticesinde bir kimlik almaktayız. Bu kimliği artık kendi bünyemizde taşımaktayız. Browser artık ne zaman aynı yere gelirse bu kimlik ile gelmektedir. Uygulama da bu kimliğe bakınca kendi ürettiği kimlik olduğunu görerek doğrulayabilmektedir. Eğer aktif bir kimlik ise geçişe izin vererek onaylamaktadır.

Peki Üretilen Cookie Sunucu Tarafında Nerede Saklanmaktadır? - Cookie ve Session Hakkında

Burada bir web uygulaması, bir kullanıcı ve diğer sistemler bulunmaktadır. Sağ tarafta cookie geldikten sonra web uygulamasının bunu kontrol etmesi gerekmektedir.



Cookie Kısaca Ne Demek ?

Bir anahtar veya Kimlik.

Session Ne Demek ?

Bu kimliğe dair bilgilerin yazıldığı ve tutulduğu bilgi topluluğudur. Genellikle sunucu tarafındadır. Bu iki kavram birbiriyle ilişkilidir ancak aynı şey değildir. Cookie sadece sunucu tarafında o user'ın kim olduğunu anlamamızı sağlayan bir şeydir.

Bu Session'u uygulamalar Nerede Tutar?

Yöntem-1)

Diskte kendi üzerinde tutabilir. Diskte tutulduğu zaman ortaya şöyle problemler çıkabilmektedir. Kullanıcı her geldiğinde cookie değeri alınıp diskteki karşılığına bakılacaktır. Bu adımlar da diskte ip işlemlerine sebep olmaktadır. Yani request ve response döngüsü yavaşlar. Sürekli bir yazma silme işlemi gerçekleşir.

Bir diğer durum da şöyledir, yukarıdaki görselde de görebileceğiniz üzere web application'ın çalıştığı uygulama sayısı birden ikiye çıkarsa ve ön tarafa bu gelen request'leri dağıtmak üzere sorumlu bir reverse proxy konursa bu durumda sizin session'ları senkronize etmeniz gerekecektir. Yani diskte tutmak çok mantıklı ve doğru bir yöntem değildir.

Yöntem-2)

Peki bir database'de Saklarsak Ne Olur?

Bu durumda bir cookie geldiğinde veritabanında sorgulama yaparak varolup varolmadığını tespit edebiliriz ve doğru kişiye session'daki tüm bilgileri database'den alabiliriz. Aynı zamanda az önce bahsettiğimiz senkronizasyon problemi de ortadan kalkacaktır. Çünkü oradaki web uygulaması da gidip database'den veri alabilecektir. Günümüzde en çok karşılaştığımız yapı bu şekilde database'de tutulmasıdır. Ancak bunun da olumsuz yanları bulunmaktadır. Yük ve performans konusu devreye girmektedir. O yüzden 3. bir yöntem bulunmaktadır. Redis.

Yöntem-3)

Redis Nedir?

Redis; hiçbir diske dokunmayan, sadece bir işletim sistemi aracılığıyla hafızada key-value şeklinde veri tutan bir servistir. Memory'de tutulduğu için de çok hızlı çalışmaktadır. Eğer session ile ilgili oluşturduğunuz bilgileri tutup kontrol ettiğiniz yeri Redis servisi yaparsanız, hem çok hızlı bir şekilde işlemlerinizi gerçekleştirebilirsiniz hem de herhangi bir veri kaybı olduğunda yaşanabilecek en kötü ihtimal user'ın 302 kodu almasıdır. Yeniden login olarak kolaylıkla cookie'sini yenileyebilir.

Yöntem-4:

4. Yöntem de session bilgisinin client'ta tutulmasıdır. Buna da Cookie Based Session denilmektedir. Normalde user'a cookie bilgisi verilmekteydi ancak burada onun yerine şöyle bir yaklaşım izlenir;

Burada bir json objesi oluşturulur. Bir IV değerine random bir değer atanır ve kullanıcıdan gelen bilgiler Json objesine eklenir. Sonuç olarak bir Session Json'u oluşturulur. Daha sonra bir Checksum olmalı. Burada Symetric Encryption yapılacağı için IV değeri bize lazım olacaktır. Daha sonra session data'sının tamamını şifreliyoruz. Bunu şifrelerken kullandığımız şey sunucu tarafında config dosyasında bulunan SYMC ENC SUPER SECRET KEY değerini kullanırız. Daha sonra da verimizde herhangi bir değişiklik olmadığından emin olmak için HMAC ile objenin kendisi ve KEY değerini birleştirerek bir Signature oluştururuz. Daha sonra da bu değeri Base64 ile encode ederiz. Ve artık user'a bu değeri veririz. Session'ı artık kullanıcı taşıyacaktır.

Bize gelen kullanıcı adı ve parolayı doğruladık, bu user ile ilgili bilgileri toplayarak obje oluşturduk. Bunu simetrik şifreleme ile şifreledik. Bunu yaparken de ayarladığımız encryption key'ini kullandık. Daha sonra da hmac ile signature elde ettik. Bunu da bir paket olarak bir cüzdan oluşturmuş olduk. Artık bunun içeriği açılmamakta ve

değiştirilememektedir. Nihayetinde bu değer ile size geldiğinde anahtar sizde olduğu için açıp tüm session'ı görebilmektesiniz.

```
SUNUCU TARAFINDA CONFİG DOSYASINDA BULUNAN SYMC ENC SUPER SECRET KEY ...!!!
```

```
username=mehmet&password=twitch
```

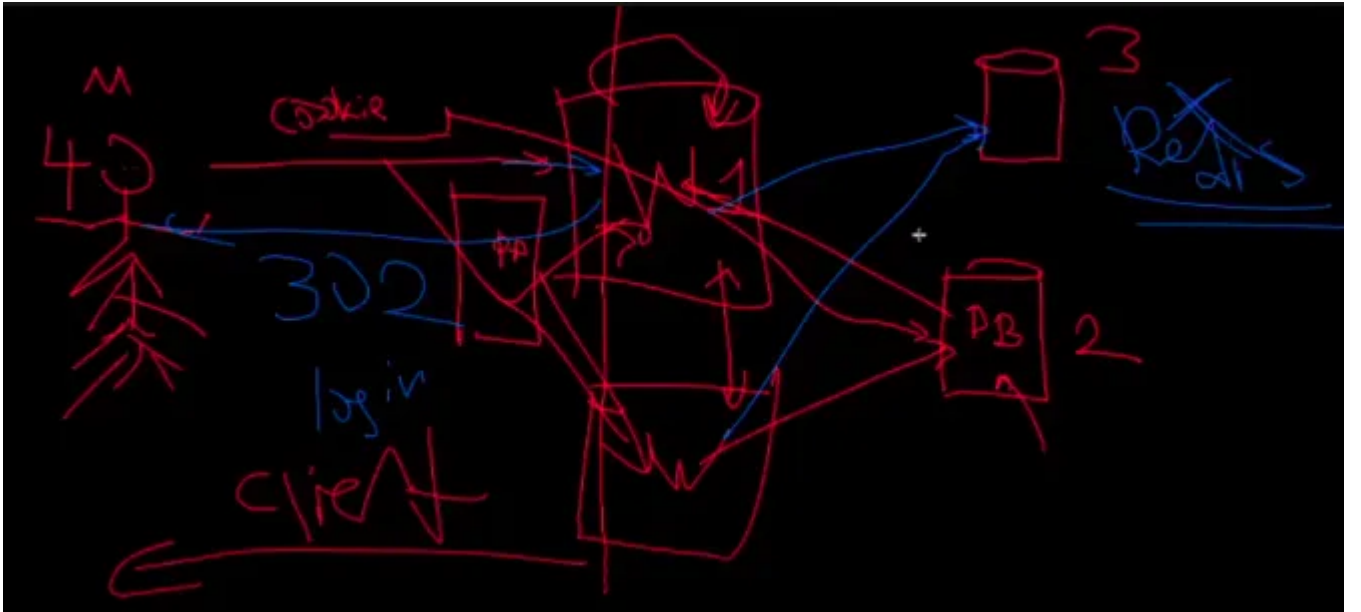
```
HTTP 302 OK
```

```
Location: mdisec.com/dashboard
```

```
Set-Cookie: {
```

```
'IV': 'RANDOM_DEGER',
```

```
'session_data': ENC(['email', 'user_id'])} | CHECKSUM --SIGNATURE --> BASE64ENCODE
```



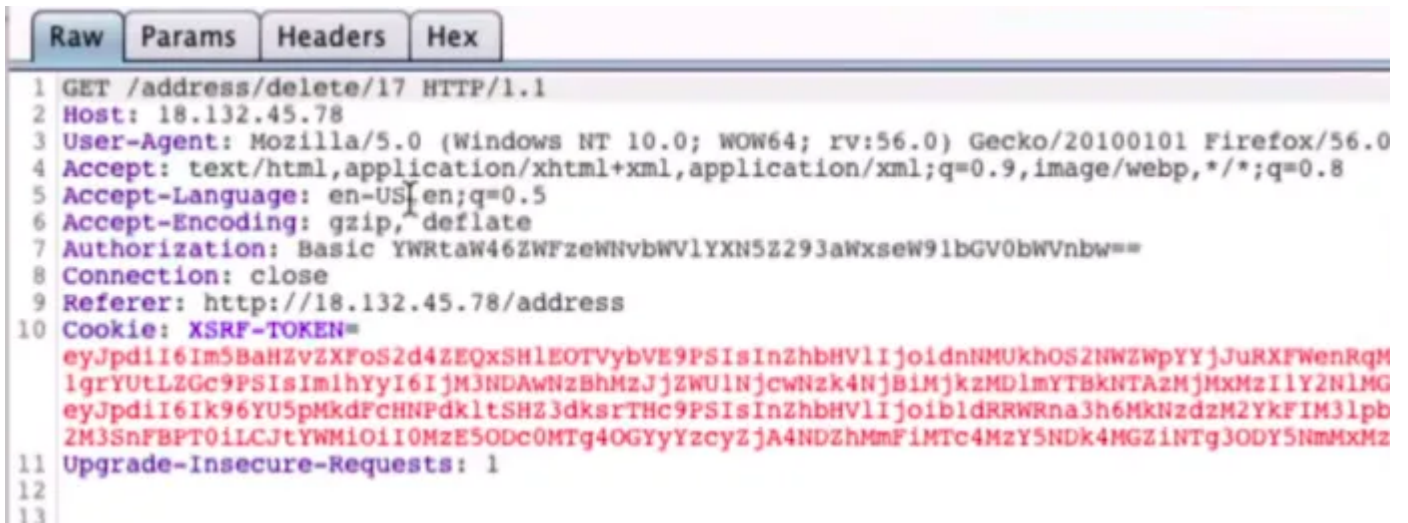
Burada okumanızı önereceğimiz bir ek kaynak da bırakalım:

<https://guides.rubyonrails.org/security.html>

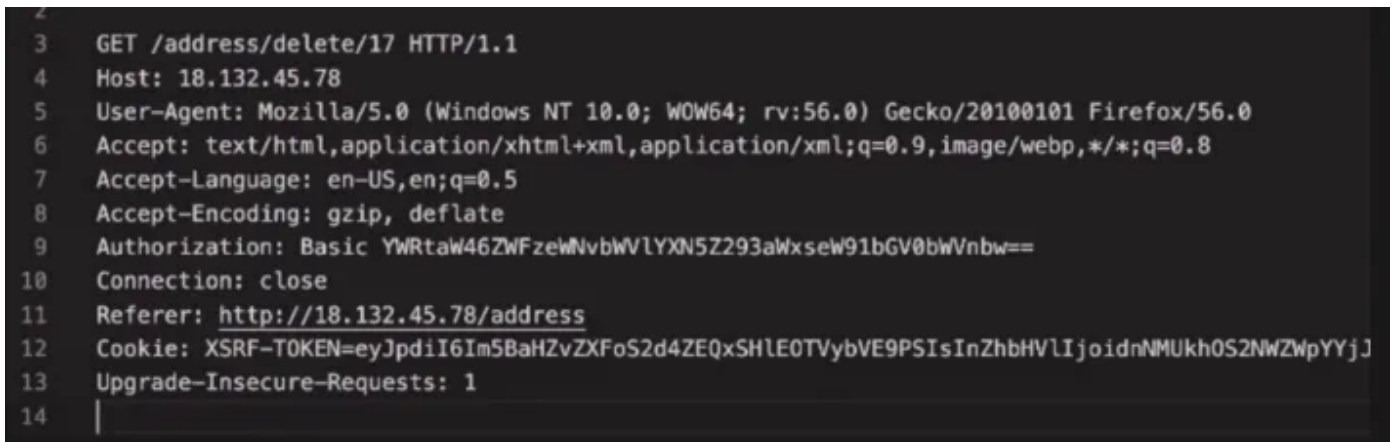
Peki tüm bu anlatılan konular ile CSRF nerede birleşmektedir?

CSRF Hakkında

Örnek bir delete request'i;



burpsuite



vscode ile inceleme

Burada webb uygulaması bu request'i user'ın isteyerek mi gönderdiğini yoksa farketmeden mi gönderdiğini anlamak zorunda.

```
1. TAB
18.132.45.78

2. TAB
www.hacker.com
```

```
<html>
  </img>
  <h1> Bu siteye giren 1M'inci kişi oldunuz... </h1>
</html>
```

Burada browser img tag'ini görünce mecburen buradaki adrese HTTP request'i gönderecektir. Hayırlayacağınız üzere bir request'i gönderirken cookie "domain"+"protocol"+"port" yapısına göre eklenmekteydi. Yani browser bu resmi yükleyebilmek için şöyle bir GET request'i üretecektir. Yani yukarıdaki request'in aynısını üretecektir. Çünkü cookie'ler bu adresle eşleşmektedir ve bundan dolayı eklemek zorundadır. Ancak hacker.com'a giren kişi böyle bir request gönderdiğinin farkında değildir. Dolayısıyla web uygulaması gelen bu request'e bakarak cookie değerinin eşleştiğini görür ve işlemi gerçekleştirir. Dolayısıyla Silme işlemi tamamlanmış olur.

```
2
3 GET /address/delete/17 HTTP/1.1
4 Host: 18.132.45.78
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0
6 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
7 Accept-Language: en-US,en;q=0.5
8 Accept-Encoding: gzip, deflate
9 Authorization: Basic YWRtaW46ZWZlZWVlYXN5Z293aWxseW91bGV0bWVnbw==
10 Connection: close
11 Referer: http://18.132.45.78/address
12 Cookie: XSRF-TOKEN=eyJpdjI6Im5BaHZvZXFoS2d4ZEQxSHlE0TVybVE9PSIsInZhbnVlIjoiaidnNMUkh0S2NWZwpY
13 Upgrade-Insecure-Requests: 1
14
```

CSRF'in ortaya çıkış hikayesi temelde bu şekildedir. Cross Site Request Forgery, yani siteler arası istek sahteciliği şeklinde türkçeye çevirebiliriz.

CSRF'in engellenmesi için bu request'in user tarafından bilinçli bir şekilde mi yoksa farkında olmadan mı üretildiğini tespit etmek gerekir. Bu tespitin yapılabilmesi için de hassas işlemlerde csrf_token gibi bir değerın sunucuya gönderilmesi gerekmektedir.

```
<p><b>#1 - asdasd</b></p>
<div class="row">
  <div class="col-sm-9">
    asdasd
  </div>
  <div class="col-sm-3">
    <a href="http://18.132.45.78/address/delete/17">Delete</a>
  </div>
</div>
<hr>
</div>
```

Raw	Params	Headers	Hex
1	POST /address HTTP/1.1		
2	Host: 18.132.45.78		
3	User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0		
4	Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8		
5	Accept-Language: en-US,en;q=0.5		
6	Accept-Encoding: gzip, deflate		
7	Content-Type: application/x-www-form-urlencoded		
8	Content-Length: 74		
9	Origin: http://18.132.45.78		
10	Authorization: Basic YWRtaW46ZWZzeWVnbWVlYXN5Z293aWxseW91bGV0bWVnbWw=		
11	Connection: close		
12	Referer: http://18.132.45.78/address		
13	Cookie: XSRF-TOKEN=		
14	eyJpdil6Im5BaH2vZXFos2d4ZEQxSH1EOTVybVE9PSIsInZhbHVlIjoiaidnNHUkhOS2hWZWPYYjJuXXFWenRqHzRDBW		
15	lqRYUTLZGc9PSIsImhYyI6IjMjNDANzBhMzJjZWUinJcWVnZk4NjB1MjZkMDlmYTgkNTAzmjYkMzIiY2N1MGYxZmQ		
16	eyJpdil6Ik96YU5pMkdFcHNpdktSHZ3dksrTHc9PSIsInZhbHVlIjoibldkRRWRna3h6MkNzdzM2YkFIM3lpbE04dG		
17	2M3SnFBPT01LCJTYWMI0110MzE5ODc0MTg4OGYyYzcyZjA4NDZhMmF1MTE4MzY5NDk4MGZlNTg3ODY5Nm9kMzU2MmM		
18	Upgrade-Insecure-Requests: 1		
19			
20	token=BapFFvHAe8riG3sk8ZFJGcgglVtcxPkOP4cUSGw&name=qeqwe&content=qeqwe		

```

</div>
<div class="col-sm-3">
  <a href="http://18.132.45.78/address/delete/17">Delete</a>
</div>
</div>
<hr>
</div>

col-sm-6">
Add New Address</h3>
You can add new delivery location.</p>
<input type="POST" action="http://18.132.45.78/address">
<input type="hidden" name="_token" value="BapFFvHAe8riG3sk8ZFJGcgglVtcxPkOP4cUSGw">
<label>Name</label>
<input type="text" name="name" class="form-control">
<label>Address</label>
<textarea name="content" rows="5" class="form-control"></textarea>
<br>
<button type="submit" class="btn btn-default">Submit</button>
</div>

```

Burada token değeri sadece kullanıcının sayfasına eklenir. Hacker kendi sitesine bu token değerini ekleyemediği için de artık işlemi gerçekleştirememektedir.

Peki REST API olduğunda ne olacak ?

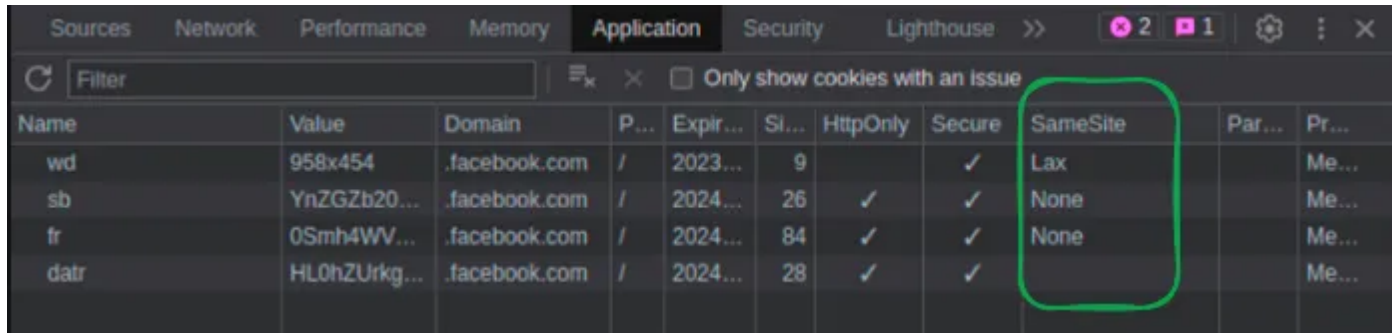
Burada CSRF_TOKEN olmamasına rağmen CSRF zafiyeti yoktur. Çünkü browser request'i hacker.com'a gönderirken cookie'leri otomatik ekliyor. O yüzden hacker.com, header'daki alanlara erişemez. Authorization kısmına yazması gereken bilgiye artık sahip değildir.

```
3 REST API
4
5 POST /address/delete/ HTTP/1.1
6 Host: api.mdisec.com
7 User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0
8 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
9 Accept-Language: en-US,en;q=0.5
10 Accept-Encoding: gzip, deflate
11 Authorization: Basic YWRtaW46ZWZewNvbWVLYXN5Z293aWxseW91bGV0bWVnbw==
12 Connection: close
13 Origin: http://mdisec.com/
14
15 {'id':'17'}
```

SameSite Cookie

Artık browser'larda şöyle bir yapı bulunmaktadır.

Set-Cookie: session=dhdhdfhdhd; expires=Fri, 24-Apr-2024 19:50:52 GMT; SameSite



Name	Value	Domain	P...	Expir...	Sl...	HttpOnly	Secure	SameSite	Par...	Pr...
wd	958x454	.facebook.com	/	2023...	9		✓	Lax		Me...
sb	YnZGZb20...	.facebook.com	/	2024...	26	✓	✓	None		Me...
fr	OSmh4WV...	.facebook.com	/	2024...	84	✓	✓	None		Me...
datr	HL0hZUrk...	.facebook.com	/	2024...	28	✓	✓	None		Me...

SameSite ise şu işe yarar;

Request'i gönderen browser'dır. Facebook sitesinden bir set-cookie geldiğini ve kaydettiğimizi düşünelim. Hacker.com'a girdiğimizde html içeriğinde eer bir img ya da başka bir tag ile facebook sitesine request isteği göndermemizi isterse browser bunu yapmak zorundadır. Bunu yaparken de otomatik olarak cookie'yi eklemektedir. Buradan da csrf zafiyeti doğmaktadır. Artık SameSite ile browser'a "bu cookie'yi sadece cookie'yi set eden siteye gelirken eklemelisin" diye bir kural tanımlanmaktadır. Yani bu cookie hacker.com'dan çıkan http request'ine eklenmeyecektir. Başka bir siteye girince önceki siteye request isteğinde bulunsanız bile artık browser bunu engelleyecektir.

Kaynaklar:

1. <https://www.youtube.com/watch?v=CKHai0OW6BY> (Web Security 101 0x03 | Session'ı ve CSRF Zafiyetini Anlamak & SameSite Cookie Önlemi)
2. <https://afteracademy.com/blog/what-is-a-tcp-3-way-handshake-process/>
3. <https://guides.rubyonrails.org/security.html>