Web Güvenliğinde Otomasyon

Ferruh Mavituna, Ağustos 2009, WGT E-Dergi 1. Sayı

Yeni gelen teknolojilerin bir çoğu ilk başlarda çok güvensiz olur. Kimsenin firewall kullanmadığı, kimsenin kablosuz ağına şifre koymadığı, bir exploit' in internetin 1/4' ünü etkilediği zamanları görüp geçirdik. Aynı bunun gibi her üç sitenin ikisinde ***SQL Injection, Cross-site Scripting*** olduğu günleri de geride bıraktık diyebiliriz.

Tüm ağlar, işletim sistemleri bir gecede güvenli olmadı ama git gide insanların bilinçlenmesi , "firewall", "otomatik güncelleme" gibi şeylerin standart hale gelmesi ile genel olarak çok daha güvenli sistemler ile karşılaşmaya başladık. Tabii ki bu sistemlerin basit saldırılara karşı güvenli olması tüm güvenlik açıklarının yok olduğu anlamına da gelmiyor, hala güvenlik açıkları var ama daha zor yerlerdeler.

Web de benzer bir geçiş sürecinden nasibini aldı. Daha güvenli web dilleri ortaya çıktı, üç sene önce web geliştiricileri XSS (*Cross-site Scripting*)'in [1] ne olduğunu bilmiyorken bugün web geliştiricisi olmayan kişiler bile en azından bu konuda bir fikir sahibiler. Tabii ki bu bilinçlenmeye katkıda bulunan tüm XSS ve SQL Injection solucanlarının da hakkını vermemiz lazım, onlar sayesinde bu tip sorunlar medayada kendilerine yer buldular [2].

Güvenlik hakkındaki bu genel bilinçlenme bir çok güvenlik açığının oluşmasına engel olmaya başladı. Artık hiç bir sitenin login formunda SQL Injection bulunmuyor ama hala bir çok web uygulamasında "Nümerik" değerler bekleyen SQL cümleciklerinde data tip kontrolü yerine sadece tek tırnaklardan kaçıldığını görebilirsiniz, mesela şu iki SQL cümleciğine bakalım:

1. SELECT * FROM users WHERE name="' + SQLSafe(Request("username")) + '"
2. SELECT * FROM users WHERE name=" + SQLSafe(Request("userid"))

Burada kullanılan SQLSafe() fonksiyonuna bakalım. Programcı MS SQL Server kullandığından tek tırnakları çift tek tırnağa çevirerek SQL Injection' a karşı kendini koruyor:

```
Function SQLSafe(input)
  input = Replace("'","''")
  SQLSafe = input
End Function
```

Bu korunma birinci SQL Query örneğinde güvenli olsa da ikincisinde güvenli değil çünkü SQL Query nümerik olduğundan tırnak içerisinde yazılmamış. Dolayısıyla zaten SQL Injection için tırnak kullanmaya gerek yok. Bunun anlamı bir saldırgan tek tırnak kullanmadan başarılı şekilde bir SQL Injection saldırısı yapabilir.

Burada hala bir güvenlik açığı var ama bulması ve exploit etmesi **" ' OR '1'='1 "** yazmaktan daha zor.

Güvenlik açısından baktığımızda üç tip site görebiliriz:

1. Güvenli Siteler
2. Güvensiz Siteler
3. Güvenli olduğunu zanneden siteler

Güvenli olduğunu zannetme genelde şu nedenlerden dolayı oluşur:

1. Web geliştiricisinin güvenlik açığını **gerçekten anlamamış** olması ama hasbel kader ne olduğunu bilmesi.
2. Geliştiricinin potansiyel bir güvenlik açığını görmüş olması ama exploit **edilemeyeceğini düşünmesi**, bunun ana nedeni kendisinin bu açığı exploit edememiş olmasıdır. O edemediyse kimse edemez (!), değil mi?
3. Kullandıkları web dilinin ya da framework' ünün onlar için tüm güvenlik işlerini hallettiğini sanması. *Bu SSL' in tüm güvenlik dertlerinizi çözeceğine inanmanız kadar komik bir şey.*

Bunun harici kriptoloji, WAF (*Web Uygulaması Firewall'ı*) vs. gibi şeylerin sistemlerini bir gecede güvenli yapacağına inanan kişiler de var, onlar başka bir yazı konusu...

Rakamlar ile Web Uygulaması Testi

Hata mesajlarını görememek bir saldırganın başına gelebilecek en kötü şeylerden biridir ve artık aklı başında hiç bir site hata mesajlarını göstermiyor. Bu mesajlara erişemeden bir sistemin güvenliğini test etmek bu mesajlara erişerek ve sistemin alt yapısını izleme imkanı olarak (*white-box testing*) test etmekten çok daha farklı bir iş.

Tam anlamı [3] ile bir SQL Injection açıklığını tespit etmek istersek, şu saldırıları denemeliyiz:

. **Filtre geçmek:** 2 tip saldırı (*Mesela boşluk kabul edilmiyorsa tüm istekleri bir de boşlukları "/**/" ile değiştirerek göndermek gerekli*)
. **SQL cümleciklerini önceden bitirmek için:** 2 tip saldırı (mesela "--" ve "")
. **Gruplama içeren SQL Cümlecikleri için:** 4 tip saldırı (mesela: "","(","((",")))" gibi...)
. **String, Integer, AND / OR vs. kombinasyonları:** 2 tip
. **Injection yerine göre:** 3 tip

Sonuç: 2 * 2 * 4 * 2 * 3 = **96 HTTP isteği**

Bunun anlamı sitedeki her dinamik girdi için **96** farklı deneme yapmazsanız sistemi gerçekten yeterli şekilde test ettiğinizden emin olmazsınız demek. Vasat bir sitede 50 dinamik parametre görebilirsiniz, eğer karşıdaki veritabanını da bilmiyorsanız, diğer veritabanlarına uygun da denemeler yapmanız lazım, yani **96 * 3 * 50 = 14.400 istek**. Teorik olarak hala mükemmel bir SQL Injection testi yapamadınız, hala acayip ve tuhaf olasılıklar var [4].

Unutmamak lazım 15.000 istekten sonra hala çözmemiz gereken başka dertler var, XSS var, CSRF var, LFI var, RFI var ve daha bir çok başka abuk subuk kısaltması olan ve bulunması gereken güvenlik açığı var.

Mantıklı bir güvenlik uzmanı gidip eliyle 15.000 deneme yapmaz, genelde onun yerine **açık olabilecek** yerlere belli olasılıklar ile odaklanır ve HTTP cevaplarını en verimli şekilde analiz edip ona göre oralarda en çok tutma olasılığı olan açıkları dener.

SQL Injection, XSS, RFI, LFI gibi mantıksal sorunlara dayanmayan diğer açıklar da SQL Injection testleri kadar olmasa da çok sayıda deneme gerektiriyor.

Burada üç genel çözüm var:

1. Güvenli yazılım geliştirmek
2. White-Box (*açık kutu, sisteme erişerek*) güvenlik testi yapmak [5].
3. Testleri otomatikleştirmek

Aslında bu üçünü birbirlerinden ayırmak yanlış olur nitekim bunların hepsini en verimli oranlarda yaparak **(vakit - para) - güvenlik** üçlüsünde güzel bir oranı yakalayabilirsiniz.

Web güvenliğinde otomatik olarak tespit edilemeyecek açıklar var -en azından yapay zeka sistemleri gelişene kadar-, ama onun harici otomatik olarak tespit edilebilecek de bir çok ciddi sorun var. Güvenlik testlerinin otomatikleştirilebilecek kısımları **otomatikleştirilmeli** ve güvenlik uzmanları da **değerli vakit ve beyinlerini** daha çok otomatik olarak tespit edilemeyecek açıkları tespit etmek için kullanmalılar.

Buradaki en kritik noktalardan biri otomatize eden aracın verimliliğidir. Web uygulamalarının komplike ve değişken yapısı, Web 2.0, AJAX ve 3rd parti yazılım çılgınlığı sayesinde her gün daha da komplike hale geliyor. Bu durumda araçlar da gereken verimi veremeyebiliyorlar, güvenlik testi yapan kişi aracın tam olarak neyi yapıp neyi yapamadığını bilmediğinden dolayı ilgili araçtansa kendi altıncı hissine daha çok güveniyor.

Bu maalesef güvenlik testi otomasyonunda uzunca bir süre daha çözülemeyek konulardan biri gibi. Muhtemel ki yakında daha çok güvenlik testi yapan kişi ile otomasyonu yapan aracın bire bir ilişkisinin yükseldiği araçlar göreceğiz.

Hatta "kullanıcı haklarına" ilişkin açıkları anlayabilen, siz siteyi gezerken girdiğiniz bir şifrenin sitenin başka bir yerinde bir hidden input içerisinde base64 ile encode edilmiş olduğunu tespit edebilen, çok adımlı formların adımlarını anlayabilip, adımları geçmeyi deneyen araçlar görebiliriz, hatta görmeliyiz. Aksi takdirde kendimizi yeni açıkların da listeye eklenmesi ile otomasyon olmadan içinden çıkılması mümkün olmayan bir yerde bulacağız.

[1] 2003'te Drupal'in geliştiricilerine bir XSS açığı bildirdiğimde, bana "XSS nedir?", "Sitede Javascript çalışması neden zararlı olsun ki?" gibi bir e-mail atmışlardı.
[2] Burayı okur okumaz insanlara CSRF'nin ne kadar önemli bir açık olduğunu anlatma amaçlı bir solucan yazmaya karar vermediniz değil mi?
[3] Bir parametreye tek tırnak koyduğunuzda bir SQL Injection hatası almıyorsanız bu o sitenin güvenli olduğuna gelmez.
[4] Mesela farklı bir sistemde sizin girdiniz ile her akşam çalışan bir SQL Query'sinde SQL Injection varsa?
[5] Aslında bu da çok süper bir çözüm değil gerçek ve büyük bir sistem ile karşılaştığınız da kaynak kod güvenlik analizlerinin aylarca süreceğiniz görebilirsiniz.