JavaScript Kütüphaneleri için Girdi Doğrulama Analizi

Ekincan Ufuktepe[1] Tuğkan Tuğlular[2]

[1,2]İzmir Yüksek Teknoloji Enstitüsü, Urla, İzmir

[1]ekincanufuktepe@iyte.edu.tr [2]tugkantuglular@iyte.edu.tr

Özet. Bugün artık mobil ve web temelli yazılımlar günlük hayatın bir parçası olmuştur. Bu yazılımlar içinde JavaScript kütüphanelerinin kullanımı da son yıllarda önemli artış göstermiştir. Bu kütüphaneler sağladıkları uygulama programlama arayüzleri ile daha ziyade söz verdikleri işlevleri yerine getirmekte ancak beklenmeyen girdilere karşı dayanıklı bir yapı sunamamaktadır. Bu çalışmada mobil ve web temelli yazılımlarda yoğun olarak kullanılmakta olan beş JavaScript kütüphanesine ait işlevlerin aldığı parametreler ile kullandıkları global değişkenler üzerinde doğrulama yapıp yapmadıkları analiz edilmiştir. Bunun için bir girdi doğrulama modeli ortaya konmuştur. Bu model üzerinde geliştirilen algoritma ile JavaScript programları için tip analiz yapan TAJS yazılımı genişletilmiş ve beş JavaScript kütüphanesine uygulanmış ve elde edilen sonuçlar paylaşılmıştır.

Anahtar Kelimeler: girdi doğrulama, dayanıklılık, yazılım kütüphaneleri, JavaScript.

Abstract. Nowadays, mobile and web based software has been an integral part of our lives. In recent years, there has been an increase in usage of JavaScript libraries in those kind of software. Although these JavaScript libraries fulfill the functions they have promised with respect to the application program interfaces they provide, they are not robust against unexpected inputs. In this study, the parameters and global variables of functions in the five selected JavaScript libraries that are frequently used in mobile and web based software are analyzed for input validation. For this purpose, an input validation model has been proposed. Based on this model, a tool called TAJS that performs a type analysis on JavaScript programs has been extended with a proposed algorithm. The resulting tool is executed on five JavaScript libraries and obtained results are shared.

Keywords: input validation, robustness, software libraries, JavaScript.

1 Giriş

Yazılım kalite parametrelerinden biri olan dayanıklılık (*İng. robustness*) IEEE yazılım mühendisliği terimleri sözlüğü [1] tarafından "bir sistem veya bileşenin geçersiz girdiler veya zorlu ortam koşulları altında ne derecede doğru olarak çalıştığı"

olarak tanımlanmıştır. Dayanıklı bir sistem veya bileşen değişken ortam koşulları altında kendisinden beklenen görevleri yerine getirmeye devam etmektedir. Bu kavram zaman zaman güvenilirlik (*İng. dependability*) ile karıştırılmaktadır. Güvenilirlik; inanılırlık (*İng. reliability*), emniyet (*İng. safety*), güvenlik (*İng.* secur*ity*) gibi bir çok kalite parametresini içinde barındıran bir şemsiye kavramdır, ancak dayanıklılık güvenilirlik için ikincil bir parametredir [2]. Bu tür tanımlara rağmen geçen on yıl içinde yazılımlara ilişkin zayıflıklar (*İng. vulnerability*) içinde yetersiz girdi doğrulama önemli yer tutmaktadır. Örneğin, enjeksiyon zayıflığı OWASP (www.owasp.org) ilk on sıralamasında birinci sıradadır.

Bir uygulamanın beklenen ve beklenmeyen girdileri gerektiği gibi işleyebilme yeteneği girdi toleransı olarak tanımlanmaktadır [3]. Aynı tanım uygulama içindeki metotlar ve yazılım kütüphanesi metotları için de kullanılabilir. Girdiyi alan metot girdinin geçerli olup olmadığını inceleyebilmeli ve geçersiz girdileri fark edip gereğini yapabilmelidir. Güvenilmeyen ve hatta yapısal olmayan girdileri anlamlandırarak, onların kısıtlara uygun olduğundan emin olma süreci girdi doğrulama (*İng. input validation*) olarak tanımlanmaktadır [4].

Özellikle etkileşimli web uygulamalarına yönelik beklentilerin artması ile bu beklentileri karşılamak üzere çok sayıda JavaScript kütüphanesi geliştirilmiştir. Geliştiricilerin bu kütüphanelerin dayanıklılıklarına ilişkin değerlendirmelerde bulunmadıkları ve bu yönde yazılım geliştirmedikleri düşüncesi ile bu çalışma planlanmış ve gerçekleştirilmiştir. Bu çalışmada bir girdinin yeterli ve gerektiği kadar doğrulanıp doğrulanmadığına bakılmamıştır. Tek bir doğrulama işleminin varlığı dikkate alınmış ve en az bir kez doğrulanmış girdi için kısaca doğrulanmış girdi ifadesi kullanılmıştır.

İzleyen bölümde benzer çalışmalar incelenmiştir. 3. Bölümde izlenen yöntem ortaya konmuştur. Kullanılan TAJS yazılımı ile üzerine geliştirilen eklentiler 4. Bölümde açıklanmıştır. 5. Bölümde girdi analizine tabi tutulmuş beş JavaScript kütüphanesine ilişkin sonuçlar açıklanmış ve tartışılmıştır.

2 Benzer Çalışmalar

Bu çalışmada yazılım kütüphaneleri girdi analizine tabi tutulmuş ve analiz sonuçlarına bakarak dayanıklılıkları üzerinde yorumda bulunulmuştur. Benzer bir çalışmada Baudry et al. [5] kontrat temelli bir dayanıklılık tanımı yapmış ve buna bağlı olarak bir yazılımın beklenmeyen durum karşısında ortaya çıkacak hatayı kendisinin fark etmesi gerektiği fikrini ortaya atmıştır. Takip eden çalışmalarında [6] hatayı fark etmenin ötesinde teşhis etmeye yönelik bir yaklaşım önermişlerdir.

Yazılım dayanıklılığını incelemek için yoğun olarak kullanılan bir başka yöntem tiftikleme (*İng. fuzzing*) olarak ifade edilmektedir. Bu yaklaşımda beklenmeyen veriler çok sayıda üretilir ve yazılıma girdi olarak beslenir [7]. Beklenmeyen veri üretimi belli bir modele (örneğin, [8]) göre yapılabildiği gibi rastgele (örneğin, [9]) olarak da üretilebilir. Fuzzing ile yazılım içinde yer alan birçok zayıflığın ortaya çıkarıldığı iddia edilmiştir [10].

3 Yöntem

JavaScript kütüphaneleri için girdi doğrulama analizi üç aşamadan oluşmaktadır. İlk aşamada statik analiz ile JavaScript kodu üzerinde hedeflenen girdilere ilişkin bilgiler toplanır. İkinci aşamada hedef seçilmiş girdiler izlenerek kullanılmadan önce herhangi bir girdi doğrulamaya tabi tutulup tutulmadığı tespit edilir. Üçüncü aşamada ise girdilerin bulunduğu atak yüzeyleri üzerinden kaynak kodun hangi bölümlerine ulaşıldığı hesaplanır.

İlk aşamada JavaScript kodu üzerinde statik analiz ile fonksiyon parametreleri ve fonksiyon içinde kullanılan global değişkenler belirlenmektedir. Parametre ve global değişkenler hakkında toplanan bilgiler iki farklı ağaç veri yapısı kullanılarak saklanmaktadır. Parametreler ağacında ilk seviye alt düğümlerine fonksiyonlar, fonksiyonların alt düğümlerine ise parametreleri atanmaktadır. Global değişkenler için tasarlanan ağaç yapısında ilk seviye alt düğümlerine global değişkenler, global değişkenlerin alt düğümlerine ise çağırıldığı fonksiyonlar atanmaktadır. Bu işlemler Algoritma 1'de gösterilmektedir.

Algoritma 1. Girdi Bilgilerinin Çıkarılması (1. Aşama)

```
1. Kütüphane veri akış çıktısını çözümle.
2. Kütüphane kaynak kodunu çözümle.
3. Her fonksiyon için
4.   Her girdi (parametre ve global değişken) için
5.     Eğer girdi parametre ise
6.       Parametreler ağacına ekle.
7.     Eğer girdi global değişken ise
8.       Global değişkenler ağacına ekle.
9. Her girdi (parametre ve global değişken) için
10.     Eğer kendisi ile başka fonksiyon çağırılmış ise
11.       Çağırılan fonksiyonu ilgili ağaca ekle.
```

İkinci aşamada, parametreler tanımlandıkları fonksiyonlar üzerinden ve global değişkenler ise kullanıldıkları fonksiyonlar içinde denetlenmektedir. Bu denetleme işlemi (bakınız Algoritma 2) veri akış bilgisinden yola çıkarak parametrelerin ve global değişkenlerin kullanılmadan önce herhangi bir doğrulama işlemine tabi tutulup tutulmadığını incelemektedir. İnceleme sonucunda üç ayrı durumdan biri ilgili parametre veya global değişkene atanmaktadır. Eğer girdi (parametre veya global değişken), kullanılmadan herhangi bir doğrulama işleminden geçmiş ise kendisine "doğrulanmış" durumu atanmaktadır. Girdi kullanıldıktan sonra doğrulanmış ise veya herhangi bir doğrulama işleminden geçmemiş ise kendisine "doğrulanmamış" durumu atanmaktadır. Girdi sadece tanımlanmış ama hiçbir şekilde kullanılmamış ya da doğrulama işleminden geçmemiş ise kendisine "bilinmeyen" durumu atanmaktadır.

Algoritma 2. Girdi Doğrulama Bilgisini Çıkarılması (2. Aşama)

```
1.Kütüphane veri akış çıktısını çözümle.
2.Her fonksiyon için
3.  Her girdi (parametre ve global değişken) için
4.    Eğer girdi sorgu sırası < girdi ilk kullanım sırası
5.       fonksiyon.girdi doğrulanmış olarak güncelle.
6.    Eğer girdi sorgu sırası < girdi ilk kullanım sırası
7.       fonksiyon.girdi doğrulanmamış olarak güncelle.
8.    Eğer girdi sorgulanmamış ve kullanılmış ise
9.      fonksiyon.girdi doğrulanmamış olarak güncelle.
10.   Eğer girdi sorgulanmamış ve kullanılmamış ise
11.      fonksiyon.girdi bilinmeyen olarak güncelle.
12.Tüm global değişkenleri doğrulanmış olarak güncelle.
13.Her global değişken için,
14.   Her çağırıldığı fonksiyon içinde,
15.     Eğer en az bir tane doğrulanmamış durumu var ise
16.       Global değişkeni doğrulanmamış olarak güncelle.
```

Son olarak üçüncü aşamada girdi yüzeyleri üzerinden kaynak kodun hangi bölgelerine ulaşılabildiğine dair bilgi çıkarılmaktadır. Böylece güvenilir olmayan bir verinin kaynak kod içerisinde nerelere kadar gidebileceği tespit edilmektedir. Diğer bir deyişle, veri akış çıktısı üzerinden ulaşılabilirlilik yüzdesi (*UY*) hesaplanmaktadır (bakınız Algoritma 3). Veri akış çıktısı, fonksiyonların içerisinden hangi fonksiyonların çağırıldığına dair bilgi vermektedir. Bu bilgiler ve fonksiyon sayısı kullanılarak öncelikle bitişiklik matrisi (*B*) oluşturulmaktadır. Çizge teorisine göre, kuvveti alınan bitişiklik matrisi, alınan kuvveti kadar, hangi düğümlere ulaşabildiğine dair bilgi vermektedir. Bu bilgiyi kullanarak *N* tane düğüm içerisinden bir düğümün (fonksiyonun) en uzaktaki düğüme en uzun yolu dolanmadan en az *N-1* hamlede ulaşabildiği varsayılmaktadır. Bu nedenle bitişiklik matrisinin *1*'den *N-1*'e kadar kuvvetleri alınarak toplanmıştır. Elde edilen matris üzerinde 0'a denk gelen satırdaki düğümün sütundaki düğüme hiçbir şekilde ulaşamadığını göstermektedir. 0'dan büyük olan değer ise sütundaki düğüme ulaşılabildiğini göstermektedir.

Matris üzerindeki girdi yüzeyi olarak tanımlanan fonksiyonlar; parametreleri olan ve/veya global değişkenleri kullanan fonksiyonlar olarak seçilmiştir. Bu kriter baz alınarak, elde edilen matristeki (*G*) satırlar seçilerek sütun bazlı toplanarak, diğer bir deyişle kütüphane içerisindeki ulaşılabilir bütün noktalar elde edilmekte ve ulaşılabilirlik vektöründe (*U*) saklanmaktadır. Her fonksiyonun satır sayısı fonksiyon satır sayısı vektöründe (*FSS*) tutulmakta ve ulaşılabilirlik vektörüne bakarak ulaşılabilir satır sayısı (*USS*) ve ulaşılabilirlik yüzdesi (*UY*) hesaplanabilmektedir.

Algoritma 3. Ulaşılabilirlik Hesaplaması (3. Aşama)

```
1. FSS ← fonksiyon satır sayısı vektörü
2. B ← veri akış çıktısı üzerinden NxN bitişiklik matrisi
3. G ← 0 // NxN geçici matris
4. Her i ← 1'den (N-1)'e kadar,
```

```
5.   G = G + B^i
6. U ← 0 // 1xN ulaşılabilirlik vektörü
7. U ← G 'de girdi yüzeyi olan satıları sütun bazlı topla
8. USS ← 0 // ulaşılabilir satır sayısı
9. Her j ← 1'den N'e kadar,
10.   Eğer V1xN[j] > 0
11.     USS ← USS + FSS[j]
12.UY ← USS/kaynak kod toplam satır sayısı
```

Görsel 1'de 3 tane fonksiyon tanımlanmıştır; *fA*, *fB* ve *fC*. Bu 3 fonksiyon içinden *fA* ve *fB* birer tane parametre içermesinden dolayı girdi yüzeyleri olarak tanımlanmıştır. *fA* fonksiyonu kendi içerisinde *fC* fonksiyonunu çağırdığından *fC* fonksiyonuna ulaşılabilirliği vardır. *fB* fonksiyonu içerisinde bir tane global değişken ve bir tane parametre bulunmaktadır. Global değişken kullanılmadığı için ve sadece atama işlemi olduğundan *fB* girdi yüzeyi olarak tanımlaması için yeterli değildir. Ancak *fB* fonksiyonu parametresi olduğundan fB fonksiyonunu da girdi yüzeyi olarak tanımlamıştır.

Görsel 1. JavaScript örnek kod.

```
1.  var x;
2.  function fA(a)
3.  {
4.     fC();
5.     return a;
6.  }
7.  function fB(str)
8.  {
9.     if(str == null)
10.         x = str + "init";
11. }
12. function fC()
13. {
14.    alert("Do something");
15. }
```

Görsel 1'deki örnek JavaScript kodun bitişiklik matrisi Görsel 2'de gözükmektedir.

Görsel 2. Görsel 1'in bitişiklik matrisi

$$B = \begin{array}{c} \\ fA \\ fB \\ fC \end{array} \begin{array}{c} \begin{array}{ccc} fA & fB & fC \end{array} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

Görsel 1'de toplamda 3 fonksiyon bulunduğundan, Görsel 2'deki matrisin 2'ye kadar kuvveti alınmıştır. Sırasıyla Görsel 3'deki matrisler elde edilmiştir.

Görsel 3. Görsel 1'deki ulaşılabilirlik hesaplaması için kuvvetleri alınan matrisler ve genel kodun ulaşılabilir yerleri.

$$B^1 = \begin{matrix} \\ fA \\ fB \\ fC \end{matrix} \begin{matrix} \begin{matrix} fA & fB & fC \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{matrix}, \quad B^2 = \begin{matrix} \\ fA \\ fB \\ fC \end{matrix} \begin{matrix} \begin{matrix} fA & fB & fC \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

$$G = B^1 + B^2, \quad G = \begin{matrix} \\ fA \\ fB \\ fC \end{matrix} \begin{matrix} \begin{matrix} fA & fB & fC \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

$$U = fA \begin{matrix} \begin{matrix} fA & fB & fC \end{matrix} \\ [0 \quad 0 \quad 1] \end{matrix} + fB \begin{matrix} \begin{matrix} fA & fB & fC \end{matrix} \\ [0 \quad 0 \quad 0] \end{matrix}$$

$$U = \begin{matrix} fA & fB & fC \\ [0 & 0 & 1] \end{matrix}$$

Girdi yüzeyleri: *fA*;5, *fB*;5
Ulaşılabilir fonksiyonlar: *fC*;4

$$FSS = [fA;5 \quad fB;5 \quad fC;4]$$

USS = 5 + 5 + 4

$$UY = \frac{USS}{Toplam\ kaynak\ kod\ satır\ sayısı}, \quad UY = \frac{14}{15}$$

Girdi yüzeyleri *fA* ve *fB* tanımlandığından, *G* matrisinin sadece ilk iki satırdan *fA* ve *fB* fonksiyonları alınmış ve sütun bazlı toplanmıştır. Bu toplam sonucunda ulaşılabilirlik vektörü *U* oluşturulmuştur. *U* vektörü içerisinden 1'den büyük olan değerlerin sütunda karşılık gelen fonksiyonun satır sayısı ile girdi yüzeyleri olan fonksiyonların satır sayıları toplanmış ve ulaşılabilir satır sayısı (USS) 9 olarak hesaplanmıştır. USS toplam kaynak kod satır sayısına bölünerek ulaşılabilirlik yüzdesi (UY) %93,3 olarak hesaplanmıştır.

Görsel 1'deki koda bakıldığında fonksiyonlara toplamda giren 3 tane girdi vardır. Bunlardan bir tanesi "*x"* global değişkeni, 2 tanesi "*a*" ve "*str*" parametreleridir. Girdi doğrulama analizi çalıştırıldığında, global değişkeni "*x*" kullanılmayıp sadece değer atandığı tespit edilmiştir, bu nedenle "*x*" global değişkenine "*bilinmeyen*" durumu atanmıştır. *fA* fonksiyonuna giren "*a*" parametresi ise herhangi bir doğrulama işlemine girmeden "*a*" parametresini döndürerek kullanılmaktadır ve hiçbir doğrumala işleminden geçmemektedir. Bu nedenle "*a*" girdisine "*doğrulanmamış*" durumu atanmıştır. *fB* fonksiyonuna giren "*str*" parametresi ise fonksiyon içerisinde kullanılmıştır. Ancak kullanılmadan önce doğrulanma sürecinden geçmektedir. "*str*" girdisi kullanılmadan önce doğrulandığı için durumu "*doğrulanmış*" olarak atanmıştır.

4 Kullanılan Araçlar

TAJS, Jensen ve arkadaşları tarafından geliştirilmiş bir JavaScript statik analiz aracıdır [11]. TAJS'nin amacı geliştiricileri desteklemek adına, programlardaki hataları tespit etme ve anlama üzerine geliştirilmiştir. TAJS, DOM ve bütün ECMAScript dillerini destekleyip veri akış analizi gerçekleştiren bir araçtır. JavaScript semantiğini zengin bir örgü (*İng. lattice*) yapısıyla modeller. TAJS çıkardığı veri akışı üzerinden; çağrı çizgesi, veri tipi analizi, *eval()* fonksiyon çağırımlarının kaldırılması, kullanılmayan kod ve değişken tespiti analizi yapabilmektedir.

TAJS çağrı çizgesini *".dot"* formatında, analiz edilen kod içerisindeki fonksiyonlar arası çağrı ilişkisini yönlendirilmiş çizge ile ifade etmektedir. JavaScript dilinin dinamik doğası ve dinamik veri tipi tanımlamasından dolayı, takip edilen veri analizi stratejisi biraz farklı olmuştur. Jensen ve arkadaşları veri tipi analizi için, değişkenler üzerinde uygulanan transfer fonksiyonları (*toString()* fonksiyonu vb.) ve geliştirdikleri örgü yapısı ile atama işlemleri üzerinden veri tipi analizi yapmaktadır. Bununla beraber geliştirdikleri veri akış diyagramı üzerinde tanımladıkları ifadeler ile fonksiyonlara giren verinin okunma, atanma, sorguya girme, başka fonksiyona geçirilmesi, fonksiyonun sonucunun döndürülmesi gibi yönergelerde ifade edilmiştir.

Yapılan analizler veri akış çıktısı üzerinden gerçekleştiği için, TAJS'de yapılan geliştirmeler veri akış çıktısı çıkarma özelliği üzerinde olmuştur. TAJS belli bir fonksiyon üzerinde veri akış bilgisini çıkarırken, kendi başka bir fonksiyonu çağırdığında hangi fonksiyonun çağırıldığı bilgisi aktarılmamıştır ve kaynak kodun çağrı çizgesi bilgisinin çıkarılması işleminde ise bu kritik bir bilgidir. Bu nedenle veri akış çizgesindeki çağırılma satırına bağlı olarak, kaynak kod içerisinde denk gelen satır ile eşleştirilerek veri akış çıktısında yer almayan fonksiyon isimleri eklenmiştir. Böylece fonksiyonların veri akış çıktısı üzerinden çağrı çizgesi çıkarma işlemi çok daha kolay olmuştur. Bununla beraber TAJS aracına bildiride sunulan çalışma kapsamında iki eklenti yapılmıştır.

TAJS üzerinde yapılan ilk eklenti, veri akış özelliğini bağımsız bir şekilde çalışmasını da etkilemeyecek şekilde, ancak veri akış özelliğinden faydalanarak girdilerin doğrulanıp doğrulanmadığına dair analiz yapan bir eklenti geliştirilmiştir. Bu eklenti TAJS içerisinde herhangi başka bir eklentiyi etkilemeyecek şekilde veri akış özelliğine bağlı olarak çalışmaktadır. Girdi doğrulama analizini gerçekleştirmek için arka planda kendi geliştirdiğimiz bir veri yapısı üzerinden ilerlenmiştir. Girdi doğrulama analizi, girdilerin veri akış çıktısındaki girdilerin hareketleri incelenerek, gerek nerelerde girdi doğrulama eksikliği olduğunu ve en sonda girdi doğrulamaya dayalı istatistik vermektedir.

TAJS'nin oluşturduğu veri akış çizgeleri bazı JavaScript fonksiyonlarının tanımlanma (*İng. declaration*) yönteminden dolayı bu fonksiyonların isimlerini veri akış çizgesine aktaramamaktadır. Ancak bu fonksiyonların tanımlandığı satırlar veri akış çizgesinde içerilmektedir. Bu nedenle JavaScript kodu, veri akış çizgesi bilgilerini içeren dosya çıktısıyla beraber incelenerek, isimsiz kalmış fonksiyonların isimleri bulunmaktadır.

TAJS'nin kendi içerisinde tanımladığı ve öncesinde geliştirmiş oldukları çağrı çizgesi çıkarma özelliği bulunmaktadır. Ancak bu çalışmada biraz daha genelleştirmeye

yakınlaştırmak için JavaScript kütüphaneleri kullanılmıştır. Kütüphane yapıları ve uygulama yapıları birbirinden farklı olduğu için, kütüphane çağrı çizgesi çıkarımında farklı bir yaklaşımla, ulaşılabilirlik üzerine TAJS'ye eklenti yapılmıştır. Çağrı çizgesi görsel olarak çıkartmaktan ziyade, veri akış çıktısı üzerinde fonksiyonların yaptığı çağrılar üzerinden bitişiklik matrisi çıkarılmıştır. Bu matris ile beraber ulaşılabilirlik hesaplaması için, olası girdi yüzeylerinden kod içerisinde nerelere ulaşılabildiğine dair satır sayısı (*İng. lines of code, LOC*) ölçütü üzerinden bilgi çıkarılmaktadır.

5 Durum Çalışması

Çalışmanın deneysel sonuçları için beş adet popüler JavaScript kütüphanesi seçilmiştir. Kütüphane seçiminde özellikle DOM manipülasyonu, arayüz tasarımı ve şablon (MVC) için kullanılan kütüphanelere önem verilmiştir. Bu kütüphaneler genellikle bir uygulamanın en üst katmanında, diğer bir deyişle kullanıcı arayüzünde, kullanıldığı için kullanılan fonksiyonlar girdi yüzeyleri olarak konumlanmaktadır. Bu nedenle herhangi bir girdi üzerinden beklenmeyen veri girişi yapıldığında ilk bu kütüphane fonksiyonları beklenmedik durumlara maruz kalmaktadır. Dolayısı ile kullanılan kütüphanenin dayanıklılığı ile yazılımın dayanıklılığını etkilemektedir. İncelenen beş kütüphane hakkında detalı bilgi aşağıda verilmiştir:

1. *Rivets* kütüphanesi web uygulama tasarımında veri bağlama ve şablon oluşturma amacıyla geliştirilen, güçlü ve hafif bir kütüphanedir. İncelenen sürüm 0.80 olup, kütüphaneye http://rivetsjs.com/ adresinden erişilebilir.
2. *Mustache* kütüphanesi mustache şablon sistemi ile oluşturulmuş, sorgu ve döngüler için karışık söz dizimleri kullanmak yerine etiket kullanılmasını amacı ile geliştirilmiştir. İncelenen sürüm 2.0.0 olup, kütüphaneye https://github.com/janl/mustache.js adresinden erişilebilir.
3. *UIkit* kütüphanesi web uygulamaları için hızlı ve güçlü arayüz geliştirmek için geliştirilmiştir. İncelenen sürüm 2.20.3 olup, kütüphaneye http://getuikit.com/ adresinden erişilebilir.
4. *ChartJS* kütüphanesi geliştiriciler için verilerini görselleştirmek için geliştirilen bir kütüphanedir. İncelenen sürüm 1.0.2 olup, kütüphaneye http://www.chartjs.org/ adresinden erişilebilir.
5. *Masonry* kütüphanesi örgü şeklinde dikey düzlemde elemanları optimum şekilnde yerleştirme amacı ile geliştirilmiştir. İncelenen sürüm 3.3.0 olup, kütüphaneye http://masonry.desandro.com/ adresinden erişilebilir.

İncelenen her JavaScript kütüphanesi için Bölüm 3. Yöntem kısmında açıklanan algoritmalar kullanılarak girdi analizi gerçekleştirilmiştir. Bu analize ait sonuçlar Tablo 1'de sunulmuştur.

Tablo 1. Girdi Analiz Sonuçları

Kütüphane	Toplam Satır Sayısı	Toplam Fonksiyon	Toplam Girdi	En Az Bir Kez Doğrulanmış Girdi	En Az Bir Kez Doğrulanmış Girdi Oranı	Girdi Etki Satır Oranı
Rivets	1386	130	165	19	0,115	0,584
Mustache	602	38	58	5	0,086	0,317
Ulkit	3292	317	215	22	0,102	0,133
ChartJS	3477	299	377	32	0,085	0,840
Masonry	3142	203	211	14	0,066	0,743

Elde edilen sonuçlara bakıldığında incelenen kütüphanelerde girdilerin ortalama %83,08'inin doğrulanmadığı görülmektedir. En çok doğrulanan girdilere sahip *Rivets* kütüphanesinde bile toplam girdilerin sadece 0,115 oranında en az bir kez doğrulandığı tespit edilmiştir. Tablo 1'de görülen değerler girdilerinin beklenmeyen değerlere karşı hazırlıksız olduğunu, dolayısı ile incelenen kütüphanelerin dayanıklılıklarının düşük olduğunu göstermektedir.

Tablo 1'de ayrıca girdilerin etkidiği satır sayısının, diğer bir deyişle ulaşılabilir satır sayısının, kütüphanedeki toplam satır sayısına oranı verilmiştir. Bu da beklenmeyen girdiler karşısında etkilenebilecek satır oranına denk gelmektedir. Örneğin, *ChartJS* kütüphanesi için bu oran 0,840 olarak bulunmuştur. Aynı kütüphanenin en az bir kez doğrulanmış girdi oranının da 0,085 olduğu düşünüldüğünde bu kütüphaneyi kullanan yazılımların da beklenmeyen girdilere karşı yüksek oranda risk altında olduğu söylenebilir.

Ulaşılabilirlik, ya da ulaşılabilir satır sayısı oranı, hesaplamaları sadece ulaşılabilir fonksiyon düğüm sayısı üzerinden de yapılabilirdi. Ancak böyle bir yaklaşım sonucunda tanımlanmış tüm fonksiyonların satır sayıları eşit kabul edilmiş olacaktı. 1000 satırlık ve 10 girdiye sahip bir kütüphane kodu ile 100 satırlık ve 10 girdiye sahip bir kütüphane kodu arasındaki ayrımı yapabilmek için ulaşılabilirlik hesaplamalarını ulaşılabilir fonksiyon sayısına satır sayıları eklenerek yapılmıştır.

Ulaşılabilirlik hesaplarına bakıldığında kütüphaneden kütüphaneye farklılık gösterdiği görülmektedir. Ulaşılabilirlik değerleri düşük çıkan kütüphaneler, kendi içerisindeki fonksiyonların diğer fonksiyonlarla az ilişkili olduğu anlamına gelmektedir. Ulaşılabilirlik değerleri yüksek çıkan değerler ise kütüphane içerisinde birçok fonksiyonun diğer fonksiyonlarla bağlantılı olduğu ve girdi etki oranının yüksek olduğu anlamına gelmektedir. Herhangi bir beklenmeyen girdi veya saldırı esnasında ulaşılabilirlik değeri yüksek olan bir kütüphanede, beklenmeyen girdi veya saldırının kodun birçok bölümüne ulaşabilme ve zarar verebilme şansı artmaktadır. Ancak ulaşılabilirlik değeri küçük tutulduğunda, beklenmeyen girdi veya saldırı esnasına kodun daha derinlere inebilme ve zarar verebilme olasılığı azalmaktadır.

6 Sonuç

Bu bildiride, mobil ve web temelli yazılımlarda yoğun olarak kullanılmakta olan beş JavaScript kütüphanesine ait işlevlerin aldığı parametreler ile kullandıkları global

değişkenler üzerinde doğrulama yapıp yapmadıkları analiz edilmiştir. Bunun için bir girdi doğrulama modeli ortaya konmuştur. Bu model üzerinde geliştirilen algoritma ile JavaScript programları için tip analiz yapan TAJS yazılımı genişletilmiş ve beş JavaScript kütüphanesine uygulanmıştır. Elde edilen sonuçlar bu kütüphanelerin beklenmeyen girdilere karşı dayanıklılıklarının düşük olduğunu göstermiştir.

Kaynaklar

[1] I. S. C. Committee, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos," *CA IEEE Comput. Soc.*, 1990.

[2] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 1–17, Jan. 2013.

[3] J. H. Hayes and A. J. Offutt, "Increased software reliability through input validation analysis and testing," *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*. pp. 199–209, 1999.

[4] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, "An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012, pp. 1419–1426.

[5] B. Baudry, Y. Le Traon, and J. Jezequel, "Robustness and diagnosability of OO systems designed by contracts," *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*. pp. 272–284, 2001.

[6] Y. Le Traon, B. Baudry, and J.-M. Jézéquel, "Design by contract to improve software vigilance," *Softw. Eng. IEEE Trans.*, vol. 32, no. 8, pp. 571–586, 2006.

[7] S. Winter, C. Sarbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations," *Software Engineering (ICSE), 2011 33rd International Conference on*. pp. 51–60, 2011.

[8] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He, and Z. Li, "A Model-Based Fuzz Framework to the Security Testing of TCG Software Stack Implementations," *Multimedia Information Networking and Security, 2009. MINES '09. International Conference on*, vol. 1. pp. 149–152, 2009.

[9] P. Godefroid, "Random Testing for Security: Blackbox vs. Whitebox Fuzzing," in *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007, p. 1.

[10] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 122–131.

[11] S. Jensen, A. Møller, and P. Thiemann, "Type Analysis for JavaScript," in *Static Analysis SE - 17*, vol. 5673, J. Palsberg and Z. Su, Eds. Springer Berlin Heidelberg, 2009, pp. 238–255.