

# A Pentester's Guide to Hacking OData

Gursev Singh Kalra, Principal Consultant  
McAfee® Foundstone® Professional Services

## Table of Contents

Introduction	3
<b>OData Basics</b>	3
Accessing Feeds and Entries	3
The Service Document	4
The Service Metadata Document	5
<b>OData by Example</b>	8
The READ operation	8
The DELETE operation	9
Creating and updating Entries	10
<b>Pentesting OData</b>	12
Additional considerations	14
<b>Conclusion</b>	14
<b>Acknowledgements</b>	15
<b>About the Author</b>	15
<b>About McAfee Foundstone Professional Services</b>	15

## Introduction

The Open Data Protocol (OData<sup>1</sup>) is an open web protocol for querying and updating data. OData enables the creation of HTTP-based RESTful<sup>2</sup> data services that can be used to publish and edit resources that are identified using uniform resource identifiers (URIs) with simple HTTP messages. OData is intended to be used to expose and access information from a variety of sources including, but not limited to, relational databases, file systems, content management systems, and traditional websites. It allows a consumer to query a data source over HTTP protocol and get results back in formats like Atom, JSON, or plain XML. OData can be termed as JDBC/ODBC for the Internet.

Over the last couple of years, the number of applications that support OData has risen and so has the number of live OData services. This paper looks at OData from a penetration testing perspective and introduces various OData concepts as we progress. All examples in this white paper are based on the OData sample service<sup>3</sup> available on the official OData website.

## OData Basics

At the core of OData are Feeds, which are collections of typed Entries. Each Entry represents a structured record with a key that has a list of properties of primitive or complex types. Simple OData services may consist of just a feed. More sophisticated services can have several feeds, and, in that case, they may expose a service document that lists all the top-level feeds, so that clients can discover them and find out the addresses of each of them.

The figure below summarizes the connection between entries, feeds, and the service document.

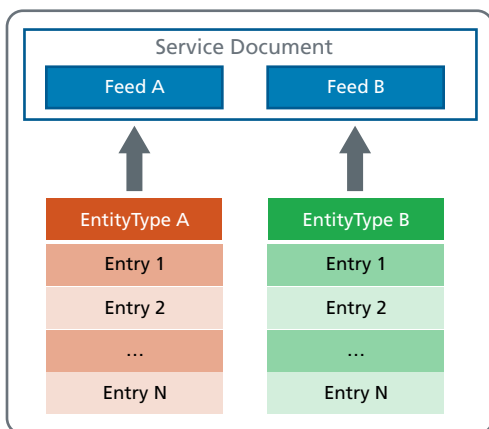


Figure 1. Relationship between Entry, Entity Type, Feed, and the Service Document.

## Accessing Feeds and Entries

The OData protocol allows URIs to identify and access individual feeds and entries. These URIs are pretty straightforward to follow; for instance, an individual Entry may look like:

```
http://localhost:32026/OData/OData.svc/Categories\(ID=0\)
```

While its feed URI would be:

```
http://localhost:32026/OData/OData.svc/Categories
```

Accessing the feed URI will result in the service responding with information concerning the entries that make up the feed, as shown in Figure 2 below:

```

localhost:32026/OData/OData.svc/Categories
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<feed xml:base="http://localhost:32026/OData/OData.svc/" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservi
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Categories</title>
  <id>http://localhost:32026/OData/OData.svc/Categories</id>
  <updated>2012-04-24T03:23:37Z</updated>
  <link rel="self" title="Categories" href="Categories" />
  <entry>
    <id>http://localhost:32026/OData/OData.svc/Categories(0)</id>
    <title type="text">Food</title>
    <updated>2012-04-24T03:23:37Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Category" href="Categories(0)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Products" type="application/atom+xml;ty
    <category term="ODataDemo.Category" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:ID m:type="Edm.Int32">0</d:ID>
        <d:Name>Food</d:Name>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>http://localhost:32026/OData/OData.svc/Categories(1)</id>
    <title type="text">Beverages</title>
    <updated>2012-04-24T03:23:37Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Category" href="Categories(1)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Products" type="application/atom+xml;ty
    <category term="ODataDemo.Category" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
  
```

Figure 2. An example Feed Retrieval.

Looking at the feed, you may notice that the `<id>` element of an Entry contains the URI that identifies a unique Entry. Based on the feed contents, restricted `<id>` values can be guessed or brute forced to gain unauthorized data access.

### The Service Document

A Service Document lists all the top level feeds exposed by the OData service. To access the Service Document, simply remove the feed portion from the URI.

For example, if the feed URI is <http://localhost:32026/OData/OData.svc/Categories>, the Service Document URI will be <http://localhost:32026/OData/OData.svc/>.

A typical Service Document should resemble the one provided by the example OData service shown in Figure 3 below:

```

view-source:localhost:32026/OData/OData.svc/
1 <?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
2 <service xml:base="http://localhost:32026/OData/OData.svc/"
3   xmlns:atom="http://www.w3.org/2005/Atom" xmlns:app="http://www.w3.org/2007/app"
4   xmlns="http://www.w3.org/2007/app">
5   <workspace>
6     <atom:title>Default</atom:title>
7     <collection href="Products">
8       <atom:title>Products</atom:title>
9     </collection>
10    <collection href="Categories">
11      <atom:title>Categories</atom:title>
12    </collection>
13    <collection href="Suppliers">
14      <atom:title>Suppliers</atom:title>
15    </collection>
16  </workspace>
17 </service>
  
```

Figure 3. An example of a Service Document.

The URI at which the service document is available is also called the Service Root URI, and it lists all the top level feeds exposed by the OData service. Individual feeds can be accessed by appending the href attribute of each collection element to the Service Root URI. Additional data access can be performed according to the feeds discovered. For example, using the service document from Figure 3 above, you can identify the following feeds:

Table 1. Various feeds.

```
http://localhost:32026/OData/OData.svc/Products
http://localhost:32026/OData/OData.svc/Categories
http://localhost:32026/OData/OData.svc/Suppliers
```

### The Service Metadata Document

The Service Metadata Document describes the different Entity Types provided by an OData service, its Service Operations (described below), the links between its resources, and several other traits of the service. You can access the Service Metadata Document for a particular OData service by appending "\$metadata" to the end of the Service Root Document URI. For example, the Service Metadata Document for the Service Root URI "<http://localhost:32026/OData/OData.svc/>" would be:

Table 2. An example of a Service Metadata Document URI.

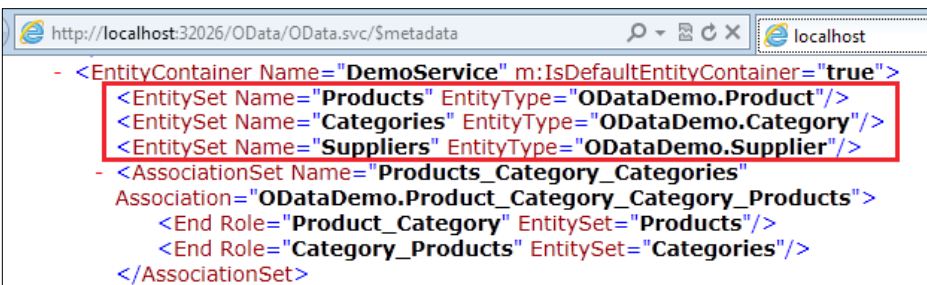
```
http://localhost:32026/OData/OData.svc/$metadata
```

The Service Metadata Document may or may not be publicly available. It uses Conceptual Schema Definition Language<sup>4</sup> to describe the OData service.

### Identifying Feeds

If you're unable to identify potential feeds using the Service Document (as shown above), you can also use the Service Metadata Document. By looking at Figure 3, it appears that feed names are always the plural of the OData Entity Type names, but this may not be the case always. The EntitySet child of the EntityContainer element provides information on the feeds exposed by an OData service. Below are the attributes of an EntitySet element:

1. Name attribute's value is the feed name seen in the Service Document.
2. EntityType attribute's value is the qualified name of the EntityType the feed exposes.



```
- <EntityContainer Name="DemoService" m:IsDefaultEntityContainer="true">
  <EntitySet Name="Products" EntityType="ODataDemo.Product"/>
  <EntitySet Name="Categories" EntityType="ODataDemo.Category"/>
  <EntitySet Name="Suppliers" EntityType="ODataDemo.Supplier"/>
  <AssociationSet Name="Products_Category_Categories"
    Association="ODataDemo.Product_Category_Category_Products">
    <End Role="Product_Category" EntitySet="Products"/>
    <End Role="Category_Products" EntitySet="Categories"/>
  </AssociationSet>
```

Figure 4. OData Feeds.

Using the information present in Figure 4, we know our Service Root URI (<http://localhost:32026/OData/OData.svc/>) and valid EntitySet names (Products, Categories and Suppliers). Putting them together, valid feed retrieval URIs would be as shown in Table 3.

Table 3. Feed URIs constructed from `EntitySet` elements.

```
http://localhost:32026/OData/OData.svc/Products  
http://localhost:32026/OData/OData.svc/Suppliers  
http://localhost:32026/OData/OData.svc/Categories
```

### Invoking Service Operations

Service Operations are simple functions exposed by an OData service whose semantics are defined by the author of the function. They typically consist of custom-written code that accepts primitive data as input parameters via GET or POST requests and return primitive types, complex types, feeds, and even a void.

Service Operations are listed as the `FunctionImport` child of the `EntityContainer` element inside the Service Metadata Document. The bordered section in the figure below shows a Service Operation definition.

```
- <AssociationSet Name="Products_Supplier_Suppliers"  
  Association="ODataDemo.Product_Supplier_Supplier_Products">  
  <End Role="Product_Supplier" EntitySet="Products"/>  
  <End Role="Supplier_Products" EntitySet="Suppliers"/>  
</AssociationSet>  
- <FunctionImport Name="GetProductsByRating" EntitySet="Products"  
  m:HttpMethod="GET" ReturnType="Collection(ODataDemo.Product)">  
  <Parameter Name="rating" Type="Edm.Int32" Mode="In"/>  
</FunctionImport>  
</EntityContainer>
```

Figure 5. A Service Operation.

`FunctionImport` has a number of attributes worth mentioning:

1. The `Name` attribute of the `FunctionImport` element is the Service Operation name.
2. The value of `EntitySet` attribute is the type of Feed returned by the Service Operation. Here, a "Products" Feed is returned when Service Operation is invoked.
3. The value of `m:HttpMethod` attribute defines the HTTP verb that should be used for invocation. Here, `GET` is required. `GET` and `POST` are the common invocation HTTP verbs.
4. The `Parameter` child element provides details on which parameters are accepted:
  - a. `Name` attribute is the name of the parameter.
  - b. `Type` attribute provides the data type expected.

The Service Operation declaration within the Service Metadata Document can be used to figure out a valid invocation URI. For instance, using the information present in Figure 5, we know our Service Root URI (<http://localhost:32026/OData/OData.svc>), a valid Service Operation (`GetProductsByRating`), its parameters (`rating`), and the type of data the parameter expects (`Int32`). Putting that all together, a valid Service Operation invocation URI would be:

Table 4. Example of a Service Operation invocation URI.

```
http://localhost:32026/OData/OData.svc/GetProductsByRating?rating=4
```

Let's try it out:

```
GET http://localhost:32026/OData/OData.svc/GetProductsByRating?rating=4 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
Accept-Encoding: gzip, deflate

1 <?xml version="1.0" encoding="utf-8" standalone="yes"?>
2 <feed xmlns:base="http://localhost:32026/OData/OData.svc/" xmlns:d="http://schemas.microsoft.com/odata/2007/08" >
3   <title type="text">GetProductsByRating</title>
4   <id>http://localhost:32026/OData/OData.svc/GetProductsByRating</id>
5   <updated>2012-05-15T14:38:35Z</updated>
6   <link rel="self" title="GetProductsByRating" href="GetProductsByRating" />
7   <entry>
8     <id>http://localhost:32026/OData/OData.svc/Products(0)</id>
9     <title type="text">Bread</title>
10    <summary type="text">Whole grain bread</summary>
11    <updated>2012-05-15T14:38:35Z</updated>
12    <author>
13      <name />
14    </author>
15    <link rel="edit" title="Product" href="Products(0)" />
16    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Category" type="application/xml" />
17    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Supplier" type="application/xml" />
18    <category term="ODataDemo.Product" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/odata/0" />
19    <content type="application/xml">
20      <m:properties>
21        <d:ID m:type="Edm.Int32">0</d:ID>
22        <d:ReleaseDate m:type="Edm.DateTime">1992-01-01T00:00:00</d:ReleaseDate>
23        <d:DiscontinuedDate m:type="Edm.DateTime" m:null="true" />

```

Figure 6. Example of a Service Operation invocation.

As you can see, we were successful in getting a valid response back from the server.

### Extracting EntityTypes

The service metadata document's `EntityType` elements describe the various `EntityTypes` exposed by the OData service. The two figures below show an example `EntityType` and a related `ComplexType`:

```
http://localhost:32026/OData/OData.svc/$metadata
- <EntityType Name="Supplier">
  - <Key>
    <PropertyRef Name="ID"/>
  </Key>
  <Property Name="ID" Nullable="false" Type="Edm.Int32"/>
  <Property Name="Name" Nullable="true" Type="Edm.String" m:FC_KeepInContent="true" m:FC_ContentKind="text" m:FC_TargetPath="SyndicationTitle"/>
  <Property Name="Address" Nullable="false" Type="ODataDemo.Address"/>
  <Property Name="Concurrency" Nullable="false" Type="Edm.Int32" ConcurrencyMode="Fixed"/>
  <NavigationProperty Name="Products" ToRole="Product_Supplier" FromRole="Supplier_Products" Relationship="ODataDemo.Product_Supplier_Supplier_Products"/>
</EntityType>
```

Figure 7. A supplier `EntityType` element.

```
http://localhost:32026/OData/OData.svc/$metadata
- <ComplexType Name="Address">
  <Property Name="Street" Nullable="true" Type="Edm.String"/>
  <Property Name="City" Nullable="true" Type="Edm.String"/>
  <Property Name="State" Nullable="true" Type="Edm.String"/>
  <Property Name="ZipCode" Nullable="true" Type="Edm.String"/>
  <Property Name="Country" Nullable="true" Type="Edm.String"/>
</ComplexType>
```

Figure 8. A complex type definition.

Using the figures above as guides, there are a number of attributes of the `EntityType` element worth mentioning:

1. The `Key` element groups together several child `PropertyRef` elements. These elements are the keys that together identify unique entries. As discussed above, OData uses named key value pairs to address unique entries.
2. The property element can represent either an OData primitive type or a complex type. Important attributes of the property element are:
  - a. *Name*—Defines name of a property.
  - b. *Nullable*—Signifies if the value is Nullable<sup>5</sup> or not.
  - c. *Type*—This can be either a primitive type (`Edm.Binary`, `Edm.String`, `Edm.DateTime`, `Edm.Int32`, and others) or a qualified name of a `ComplexType`. A `ComplexType` groups together logically related primitive types. Complex types do not have keys and cannot be instantiated on their own. If property's type attribute value is not one of the primitive types,<sup>6</sup> it should represent a qualified name of a `ComplexType` definition within the same Service Metadata Document. `Address` property in Figure 7 is a `ComplexType` property and Figure 8 shows the corresponding `ComplexType` definition from the same service metadata document.
3. It is important to point out that different `EntityTypes` can be related to each other, and there can be one-to-one or one-to-many relationship between entries.

We will visit the `NavigationProperty` element a little later in this white paper.

### OData by Example

The OData protocol supports GET, POST, PUT/MERGE, and DELETE HTTP verbs for its RESTful operations. These verbs are used to READ, CREATE, UPDATE, and DELETE records respectively.

HTTP Method	REST Equivalent
POST	CREATE
GET	READ
PUT/MERGE	UPDATE
DELETE	DELETE

Figure 9. HTTP verb and corresponding REST equivalents.

It's easy to get overwhelmed when learning a new protocol, so we'll start with the easiest of the four<sup>7</sup> types of OData operations: the "READ."

### The READ operation

Figure 10 below shows a READ operation retrieving a single OData record from an OData service.

```
GET http://localhost:32026/OData/OData.svc/Categories(0) HTTP/1.1
1 <?xml version="1.0" encoding="utf-8" standalone="yes"?>
2 <entry xml:base="http://localhost:32026/OData/OData.svc/" xmlns:d="http://schemas.microsoft.com/ado/
3   xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2
4   <id>http://localhost:32026/OData/OData.svc/Categories(0)</id>
5   <title type="text">Food</title>
6   <updated>2012-04-27T03:31:22Z</updated>
7   <author>
8     <name />
9   </author>
10  <link rel="edit" title="Category" href="Categories(0)" />
11  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Products" type="applicati
12  <content type="application/xml">
13    <m:properties>
14      <d:ID m:type="Edm.Int32">0</d:ID>
```

Figure 10. An OData READ operation with XML data.



There are a couple of items above that are worth highlighting:

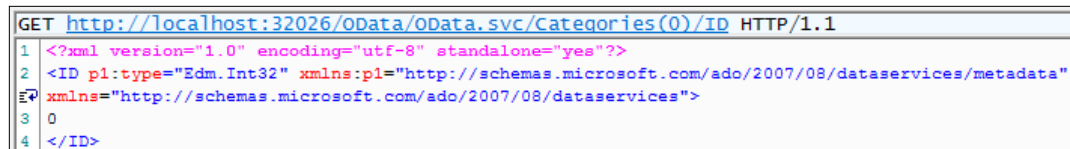
1. Category(0) on the top line that starts with a "GET" identifies a unique OData record. The "(0)" is the unique record (Entry) identifier that corresponds to primary key "0" within the back-end database. OData also allows for comma-separated named parameters. For instance, "Category(ID=0, Name='Foundstone')" identifies a unique record from a dataset that uses the named primary keys "ID" and "Name" to access data. The comma separated name value pairs must be used to access individual records if multiple keys identify a unique `EntityType`. For the special case when a single key is used to identify unique `EntityTypes`, a value can be used instead of name value pair. For example, "Category(0)" and "Category(ID=0)" will retrieve the same record.
2. The `<id>` parameter present on line 3 represents the URI that identifies a single record or Entry. This parameter is returned for every OData record accessed.
3. Lines 13 to 16 show the properties for the Entry. These properties can be either of primitive<sup>8</sup> type or complex type.

If an attacker has access to any valid Entry/Record identifier, it may be possible to guess other record identifiers to gain access to additional data.

### Extracting Individual Properties

An interesting feature of OData is the ability to request Individual Properties for an Entry. For instance, if an Entry is named "Categories" and has several properties, you can make a specific request for Individual Properties, rather than requesting all the data and having to parse it on the requesting side.

For example, if we wanted to retrieve the properties (in XML or JSON format) for the "ID" key within the "Categories" Entry, we can make the specific request shown in Figure 11.



```
GET http://localhost:32026/OData/OData.svc/Categories(0)/ID HTTP/1.1
1 <?xml version="1.0" encoding="utf-8" standalone="yes"?>
2 <ID pl:type="Edm.Int32" xmlns:pl="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
3 xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">
4 0
5 </ID>
```

Figure 11. Retrieving a single property of an Entry.

If we wanted to retrieve the raw value of the "ID" key within the "Categories" Entry, we would just append `/$value` to the end of our request as shown below in Figure 12:

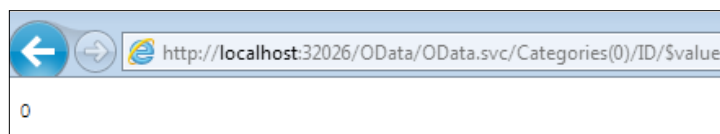


Figure 12. Retrieving the unformatted value for a property with `$value`.

### The DELETE operation

To invoke the DELETE verb, simply provide a target Entry URI to be deleted. Since DELETE has such devastating repercussions, it is important to ensure strict restrictions are put in. The GET or READ verb is equally as simple to invoke, and the target Entry URI is used in this method to define what record to access.

An example of a DELETE call is shown in the image below, followed by a request to access the deleted resource.

```
DELETE http://localhost:32026/(S(22uxzrppujxbxt4ihmycxwq))/OData/OData.svc/Categories(0)
HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: localhost:32026
HTTP/1.1 204 No Content
Server: ASP.NET Development Server/10.0.0.0
```

Figure 13. DELETE method invocation and corresponding response.

```
GET http://localhost:32026/(S(22uxzrppujxbxt4ihmycxwq))/OData/OData.svc/Categories(0)
HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: localhost:32026
HTTP/1.1 404 Not Found
Server: ASP.NET Development Server/10.0.0.0
```

Figure 14. READ attempt on a deleted Entry.

### Creating and updating Entries

CREATE and UPDATE requests allow the application interacting with OData to create or modify entries. The first important step in building one of these requests is discovering the structure of request that is specific to the application. Most of this information can be determined from the Service Metadata Document; you can also use Oyedata (<http://www.mcafee.com/us/downloads/free-tools/oyedata.aspx>), a tool that will do it for you and will also allow you to engage the OData service to retrieve and send information. Here's a quick look at what a CREATE request looks like in Oyedata:

<b>Request</b>	<input type="button" value="Send"/>
<pre>POST /Suppliers HTTP/1.1 Host: localhost Accept: application/json Content-Type: application/json  {   "ID": "2002570204",   "Name": "4w5jNIz5PK5osb",   "Concurrency": "-1256377534",   "Address": {     "Street": "x",     "City": "mfrdA",     "State": "W4b2ZKzJ4h2FpPftu",     "ZipCode": "vwhOGc9H",     "Country": "dqCdI5srpYDqA6eXuH"   } }</pre>	
←	
<b>Response</b>	
<pre>HTTP/1.1 201 Created</pre>	

Figure 15. A successful supplier CREATE operation with JSON data format.

There is quite a bit of information available around building CREATE and UPDATE requests, so it's recommended that you check out sections 2.2.7 and 4.0 (examples) in the OData documentation to learn the intricacies of how these requests are built. One nice feature is that these requests can be formatted in either JSON or XML. The XML version of the request shown in Figure 16 looks like this:

```
POST / (S(22uxzrppujxbbxt4ihmycxwg))/OData/OData.svc/Categories HTTP/1.1
Host: localhost:32026
Accept: application/atom+xml,application/atomsvc+xml,application/xml
Content-Type: application/atom+xml
Content-Length: 401

<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom" xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices">
  <content type="application/xml">
    <m:properties>
      <d:ID>-1136691363</d:ID>
      <d:Name>GL5MymUNTnDfyGh6o</d:Name>
    </m:properties>
  </content>
</entry>

HTTP/1.1 201 Created
Server: ASP.NET Development Server/10.0.0.0
```

Figure 16. A successful category CREATE operation with XML data format.

The bottom portion of Figure 16 shows the response from the server, indicating the Entry was created.

It's worth mentioning that UPDATE requests have two major differences when compared to CREATE requests:

1. OData update operations can target a complete Entry or individual properties.
2. The HTTP PUT and custom OData MERGE verbs are used to update entries when using the UPDATE method, rather than the POST which is used for CREATES.

Oyedata also supports both CREATE and UPDATE, so you can use it for whichever you'd like.

## Pentesting OData

The OData protocol does not include security specifications and suggests the implementers to use what best fits their target scenario. Since the OData protocol is based on HTTP, AtomPub, and JSON, it is subjected to the security considerations applicable to each of those technologies. Additionally, there are a number of other common penetration test cases that should be considered. The tables below provide areas of focus when approaching OData applications.

### 1. Discovery and Configuration

Service Document	<ul style="list-style-type: none"><li>• Attempt to access the service document. If accessible, enumerate the Feeds</li><li>• Construct feed URIs and try accessing them to check for unrestricted data access</li><li>• Service document accessibility must be verified as per business requirements</li></ul>
Service Metadata Document	<ul style="list-style-type: none"><li>• Verify if the Service Metadata Document is accessible. If not, obtain a copy for analysis</li><li>• Attempt to enumerate Feeds using the Service Metadata Document</li><li>• If the OData service is for private consumption, the Service Metadata Document must not be made available over the Internet. This should be an important test case for each penetration test.</li><li>• Analyze the contents of Service Metadata Document to construct various attack templates to engage the OData service</li><li>• Review <code>EntitySet</code> child elements inside the <code>EntityContainer</code> to determine the feeds exposed by the OData service and test each one of them</li></ul>
Tools	<ul style="list-style-type: none"><li>• McAfee Foundstone Oyedata can be leveraged to automate Service Metadata Document analysis and attack template creation process</li><li>• Explore if Linqpad<sup>10</sup> and other tools<sup>11</sup> meet your requirements</li></ul>
Insecure HTTP Methods	<ul style="list-style-type: none"><li>• OData relies on <code>GET</code>, <code>POST</code>, <code>DELETE</code>, and <code>PUT</code> HTTP verbs to specify the action to be performed on resources identified by the unique URIs. It is important to securely configure the web server to allow insecure methods like <code>DELETE</code> and <code>PUT</code> only for the relevant resources. Insecure configuration can possibly lead to web server compromise and website defacement.</li></ul>

### 2. OData Operations

RESTful Operations	<p>For each user privilege, identify the <code>EntityTypes</code> for which there is restricted access, no access, or access to subset of RESTful (<code>CREATE</code>, <code>READ</code>, <code>UPDATE</code>, and <code>DELETE</code>) operations and attempt the following:</p> <ol style="list-style-type: none"><li>1. Enumerate valid keys and try to access individual Entries.</li><li>2. Access individual properties for the restricted Entries to ensure granular access restrictions are in place.</li><li>3. Write operations must be attempted on Entity Sets designated as read-only. Do so by attempting <code>POST</code> requests on individual Entity Types.</li><li>4. Verify if Entries can be removed by sending <code>DELETE</code> requests on the restricted entries.</li><li>5. Update operations must be attempted at Entry, individual property, and raw value (<code>\$value</code>) levels. Do so by sending <code>PUT</code> requests.</li></ol>
Service Operations	<p>A penetration tester must test Service Operations for various injection attacks, logic flaws, authorization checks, and more</p>

### 3. Authentication, Authorization, and Session Management

General	The OData protocol does not define any scheme for authentication or authorization. Penetration testers must therefore perform comprehensive authentication, authorization, and session management tests as per the application.
HTTP Verb Tunneling	<p>To work with clients that do not support HTTP verbs like DELETE, PUT, or MERGE, OData protocol offers a technique called “verb tunneling.” In this technique, PUT, DELETE, and MERGE requests are submitted as a POST request, and an X-HTTP-Method<sup>12</sup> header specifies the actual verb that the recipient should apply to the request.</p> <p>Penetration testers must test for DELETE, PUT, or MERGE methods with “verb tunneling” to ensure that consistent access mechanisms are implemented. It is possible that direct invocation of DELETE or PUT methods may be prohibited on some resources but can be executed via X-HTTP-Method header.</p>
Navigation Properties for Additional Data Access	<p>NavigationProperty child elements of an EntityType allow navigation from one Entity to another via a relationship. The figures in “Creating and Updating Entries” section show a couple of Entity Types with NavigationProperty elements. These navigation properties can be accessed by appending the name of navigation property to a single Entry. Example URIs to retrieve related products and the product feed are provided below:</p> <ul style="list-style-type: none"><li>• <a href="http://localhost:32026/OData/OData.svc/Categories(1)/\$links/Products">http://localhost:32026/OData/OData.svc/Categories(1)/\$links/Products</a></li><li>• <a href="http://localhost:32026/OData/OData.svc/Categories(1)/Products">http://localhost:32026/OData/OData.svc/Categories(1)/Products</a></li></ul> <p>This relationship navigation mechanism can be used as a springboard to other Entry Type’s additional data, and penetration testers should test for occurrences when otherwise constrained data can be accessed.</p>
System Query Options	<p>System query options<sup>13</sup> control the amount and type of data returned by the OData service. System query option names are prefixed by “\$” character. A couple of system query options that may be of interest while performing penetration tests are summarized below:</p> <p><b>\$select</b>—\$select system query parameter can be used to retrieve subset of available properties. For example, the URI <a href="http://localhost:32026/OData/OData.svc/Categories?\$select=Name">http://localhost:32026/OData/OData.svc/Categories?\$select=Name</a> can be used to retrieve only the name property for all Entries in the Categories Feed. The \$select option can be potentially used to retrieve hidden properties by specifying the wildcard character “*” which causes all properties to be included in the returned feed. Additionally, the “*” can be used to enumerate additional property names.</p> <p>Example URI: <a href="http://localhost:32026/OData/OData.svc/Products?\$select=*">http://localhost:32026/OData/OData.svc/Products?\$select=*</a></p> <p><b>\$format</b>—\$format system query parameter’s value is used by clients to request data in a particular format. Specifying non-existing format values to this system query option has generated detailed error messages during the tests.</p>

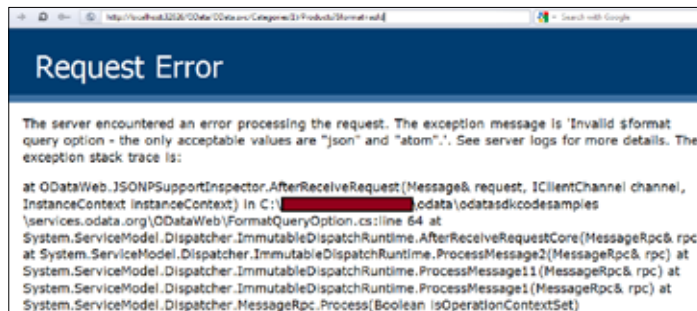


Figure 17. An error message returned when non-supported format is provided.

System query parameter values may also be tested like regular web application parameters.

Other OData system query options, \$expand, \$filter, \$orderby, \$skip, \$top, \$skiptoken, and \$inlinecount, can be applied to almost every READ request. Penetration testers are encouraged to review their invocation syntax and incorporate the applicable attack vectors during their assessments.

#### 4. Data Validation and Error Handling

---

Data Validation	An OData implementation must be tested for various data validation scenarios based on the type of back-end system it talks to in order to provide exhaustive penetration test coverage. Injection attacks and other data validation tests must also be performed as applicable.
Error Handling	Malformed JSON and XML request formats and incompatible data types should be sent to ensure proper error handling schemes exist. I have seen a couple OData services accepting invalid data during updates and creation of new Entries and then becoming unusable unless those invalid Entries are manually removed from the database.
Database Integrity Checks	Each new Entry inserted to the database has to have a unique key. Strict checks must be enforced to ensure uniqueness of keys. It was observed that the inability to maintain the uniqueness of keys leads to a corrupt database leading to a denial-of-service type of condition.

---

#### Additional considerations

The Open Data Protocol aims to provide a consistent web access mechanism for file systems, databases, CMS, and other type of data storage methods. This requires a framework that can generate a great deal of dynamic code and is customizable for different types of data sources and environments. With several types of underlying storage, interesting new types of vulnerabilities can creep in, for instance:

1. SQL injection in the dynamic code and queries that are used to interact with the underlying database.
2. File traversal vulnerabilities when files are used as data sources and file access is not appropriately sandboxed or input character filtering is not performed.
3. XPath injection when the OData implementation extracts information from XML content.
4. Framework specific vulnerabilities.

An OData implementation must be tested for various data validation scenarios based on the type of back-end system it talks to in order to provide exhaustive penetration test coverage.

#### Conclusion

As OData evolves and its usage spreads far and wide, more attacks will be discovered. Penetration testers are encouraged to review the comprehensive documentation available on the official OData website, familiarize themselves, and gain deeper understanding of this wonderful new protocol. After all, gaining a deeper understanding is the first step to securing and assessing any new technology.

## Acknowledgements

Brad Antoniewicz provided significant support by reviewing this white paper.

## About the Author

Gursev Singh Kalra serves as a principal consultant with McAfee Foundstone Professional Services, a division of McAfee. Gursev has done extensive security research on CAPTCHA schemes and implementations. He has written a Visual CAPTCHA Assessment tool, TesserCap, that was voted among the top 10 web hacks of 2011. He has identified CAPTCHA implementation vulnerabilities like CAPTCHA Re-Riding Attack, CAPTCHA Fixation and CAPTCHA Rainbow tables among others. OData security research is also one of his interests, and he has authored the OData assessment tool, Oyedata. He has also developed open source SSL Cipher enumeration tool, SSLSmart, and has spoken at a wide variety of conferences, including ToorCon, OWASP, NullCon, Infosec Southwest, and Clubhack.

## About McAfee Foundstone Professional Services

McAfee Foundstone Professional Services, a division of McAfee, offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, McAfee Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company's professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military.

## About McAfee

McAfee, a wholly owned subsidiary of Intel Corporation (NASDAQ:INTC), is the world's largest dedicated security technology company. McAfee delivers proactive and proven solutions and services that help secure systems, networks, and mobile devices around the world, allowing users to safely connect to the Internet, browse, and shop the web more securely. Backed by its unrivaled global threat intelligence, McAfee creates innovative products that empower home users, businesses, the public sector, and service providers by enabling them to prove compliance with regulations, protect data, prevent disruptions, identify vulnerabilities, and continuously monitor and improve their security. McAfee is relentlessly focused on constantly finding new ways to keep our customers safe. <http://www.mcafee.com>



2821 Mission College Boulevard  
Santa Clara, CA 95054  
888 847 8766  
www.mcafee.com

- <sup>1</sup> <http://www.odata.org/>
- <sup>2</sup> [http://en.wikipedia.org/wiki/Representational\\_state\\_transfer](http://en.wikipedia.org/wiki/Representational_state_transfer)
- <sup>3</sup> <http://www.odata.org/ecosystem#samplecode>
- <sup>4</sup> <http://msdn.microsoft.com/en-us/library/bb399292.aspx>
- <sup>5</sup> [http://msdn.microsoft.com/en-us/library/1t3y8s4s\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/1t3y8s4s(v=vs.80).aspx)
- <sup>6</sup> <http://www.odata.org/documentation/atom-format#PrimitiveTypes>
- <sup>7</sup> CREATE, READ, UPDATE, and DELETE are the four RESTful operations supported by the OData protocol.
- <sup>8</sup> <http://www.odata.org/documentation/overview#AbstractTypeSystem>
- <sup>9</sup> [http://www.odata.org/media/16352/\[ms-odata\].pdf](http://www.odata.org/media/16352/[ms-odata].pdf)
- <sup>10</sup> <http://www.linqpad.net/>
- <sup>11</sup> <http://www.odata.org/ecosystem#consumers>
- <sup>12</sup> <http://www.odata.org/documentation/operations#AdditionalInteractionModelConsiderations>
- <sup>13</sup> <http://www.odata.org/documentation/uri-conventions#SystemQueryOptions>

---

McAfee, the McAfee logo, and McAfee Foundstone are registered trademarks or trademarks of McAfee, Inc. or its subsidiaries in the United States and other countries. Other marks and brands may be claimed as the property of others. The product plans, specifications and descriptions herein are provided for information only and subject to change without notice, and are provided without warranty of any kind, express or implied. Copyright © 2012 McAfee, Inc.  
47501wp\_hacking-odata\_0812\_fnl\_ETMG