Microsoft
Global Foundation Services

SECURE USE OF CLOUD STORAGE

BLACKHAT BRIEFINGS USA 2010

AUTHOR

Grant Bugher
Lead Security Program Manager, Online Services Security and Compliance
Global Foundation Services, Microsoft Corporation

JULY 2010

VERSION 1.0 (FINAL)

Table of Contents

TABLE OF CONTENTS .... **II**

EXECUTIVE SUMMARY **1**

INTRODUCTION **1**

CLOUD STORAGE SYSTEMS **1**

MICROSOFT WINDOWS AZURE STORAGE 1

Blob Service *2*

Table Service *2*

Queue Service *2*

SQL Azure *3*

REST API *3*

.NET API *7*

AMAZON SIMPLE STORAGE SERVICE 8

Simple Storage Service (S3) *8*

SimpleDB *8*

Relational Database Service *9*

REST API *9*

SOAP API *12*

TRADITIONAL DATABASE ATTACKS **12**

SQL INJECTION 12

SQL Statements *12*

SQL Injection *13*

XML INJECTION 14

ATTACKS ON CLOUD STORAGE APPLICATIONS **16**

QUERY STRING INJECTION 16

DIRECT NODE INJECTION 17

CDATA AND XML COMMENT INJECTION 18

PERSISTENT CROSS-SITE SCRIPTING 20

ENUMERATION 20

ACCOUNT ATTACKS 24

DEFENSES AND MITIGATIONS **25**

STANDARD MITIGATIONS 25

Input Validation *25*

Encoding *25*

Windows Communication Foundation *26*

SOAP Toolkits *26*

Encryption and Multifactor Authentication *27*

INEFFECTIVE MITIGATIONS 27
APPLICATION DESIGN MITIGATIONS 28
Shared Access Signatures *28*
Gatekeeper Design Pattern *30*

CONCLUSION 31

APPENDIX A: ADDITIONAL RESOURCES 32

APPENDIX B: GLOSSARY 33

Executive Summary

Cloud storage systems like those offered by Microsoft Windows Azure and Amazon Web Services provide the ability to store large amounts of structured or unstructured data in a way that promises high levels of availability, performance, and scalability. However, just as with traditional data storage methods such as SQL-based relational databases, the interfaces to these data storage systems can exploited by an attacker to gain unauthorized access if they are not used correctly.

The query strings used by cloud providers of tabular data make use of query strings that are subject to SQL Injection-like attacks, while the XML transport interfaces of these systems are themselves subject to injection in some circumstances. In addition, cloud-based databases can still be used for old attacks like persistent cross-site scripting. Using cloud services to host public and semi-public files may introduce new information disclosure vulnerabilities. Finally, owners of applications must ensure that the application's cloud storage endpoints are adequately protected and do not allow unauthorized access by other applications or users.

Luckily for developers, modern development platforms offer mitigations that can make use of cloud services much safer. Conducting database access via frameworks like Windows Communication Foundation and SOAP toolkits can greatly reduce the opportunity for attacks, and cloud service providers themselves are beginning to offer multifactor authentication and other protections for the back-end databases. Finally, traditional defense-in-depth measures like input validation and output encoding remain as important as ever in the new world of cloud-based data.

Introduction

This paper covers background on cloud storage, an overview of database attacks, and specific examples using two major cloud storage APIs – Windows Azure Storage and Amazon S3/SimpleDB – of exploitable and non-exploitable applications. The purpose of this paper is to teach developers how to safely leverage cloud storage without creating vulnerable applications.

Cloud Storage Systems

Microsoft Windows Azure Storage

The Windows Azure platform provides a variety of options for persistent storage of application data. This paper will focus on the actual Windows Azure Storage platforms, and not on storage options such as SQL Azure and local storage on the VM. SQL Azure is a standard database platform, akin to Microsoft SQL Server, and is subject to the same attacks and mitigations as any other SQL-based database. Local storage on the VM, on the other hand, is ephemeral (it does not persist when a VM is shut down or moved) and thus unsuitable for storing anything but temporary files and caches, and thus is a less interesting target for database attacks. (Of course, a poorly-written application may still have vulnerabilities in its use of local storage and files, but these are out of scope for this paper as they do not involve cloud storage.)

When an Azure Storage Service instance is created on the Developer Portal, three service endpoints are created – one each for Blob, Table, and Queue storage. These have predictable domain names as follows:

Blob: http://*accountname*.blob.core.windows.net
Table: http://*accountname*.table.core.windows.net
Queue: http://*accountname*.queue.core.windows.net

Blob Service

Azure Blob Storage allows an application to store arbitrary blobs (binary objects), designated by a container name and a blob name. One storage account can contain one or more containers, each of which can contain one or more blobs. Blobs can be very large (up to 1 TB each), and can have associated metadata. Blobs can be distributed via the Azure Content Delivery Network (CDN), and can even be mounted as drives by services running in the Windows Azure fabric.

These blobs can either be public (in which case they can be accessed by anyone who knows the URL) or protected by shared access signatures. In the case of protected blobs, access must come through API calls that provide an appropriate authentication header, which is used by the Azure Blob Storage service to determine which rights to grant. Likewise, data definition or metadata queries come through API calls with specific, designated headers. Public blobs, on the other hand, can be accessed with any web browser.

Table Service

Azure Table Storage allows an application to store arbitrarily-formatted tabular data. Each row of Azure Table Storage may contain any number of named values. While the normal use of this service is to represent a traditional database table, with consistent columns for each row, it is actually possible for a single table to contain rows with different schemas – they are more akin to a set of entities each of which has a list of named properties. Tables can be very large (billions of rows, multiple terabytes of data.)

While Azure Table Storage superficially resembles a table in a relational database, it is important to understand that Table Storage data is not relational. The database does not recognize a concept of foreign keys, and thus does not enforce any sort of referential integrity. An application that requires these concepts will need to enforce them in application code, rather than relying on the underlying database to do it. (An Azure developer who requires relational database functionality also has the option of using Microsoft's SQL Azure Database service offering, which is a true relational database accessed with SQL rather than with Azure storage APIs.)

Tables are accessed using a Representational State Transfer (REST) API. Microsoft also provides functionality in the Azure SDK and code samples that translate more familiar ADO.NET Data Services and LINQ queries into Azure Table Storage REST calls.

Queue Service

Azure Queue Storage is generally used for communication between Azure roles and applications rather than storage of data at rest. A queue contains a list of messages that can be added to or read from; normal usage could include a Web role insance writing a message into a queue describing work to be done, and a Worker role monitoring that queue and, when messages appear, processing work requests. Queues are also accessed via a REST API.

SQL Azure

SQL Azure Database provides a true relational database service in the cloud. An application using SQL Azure can make use of most of the standard functionality expected from a full-featured relational database like Microsoft SQL Server. Tables, indexes, views, stored procedures, triggers, and constraints are fully supported; however, use of CLR (.NET) stored procedures is not supported, nor are services such as the service broker, SQL Reporting Services, and SQL Analysis Services.

Rather than using a REST API like the other Azure storage services, SQL Azure is accessed via Tabular Data Stream (TDS), the same protocol used by Microsoft SQL Server (operating over port TCP/1433.) Thus, any Microsoft SQL Server client library can also use SQL Azure, and since a SQL Azure Database looks like an ordinary SQL Server system, standard tools like SQL Server Management Studio, SQL Server Integration Studio, and BCP can be used to manage the data.

From an application security perspective, SQL Azure is identical to any other SQL-based database, and is thus out of scope for this paper.

REST API

Azure Storage uses simple REST APIs for retrieving and storing data in Blob, Table, and Queue storage.

Retrieving a Blob

Accessing an existing blob is as simple as issuing an HTTP 1.1 GET request:

```
GET http://accountname.blob.core.windows.net/containername/blobname?snapshot=datetime HTTP/1.1
```

For public blobs, this is all that is required; even the snapshot parameter is optional as most blobs are not versioned. If a blob is set private, then an authentication token must be supplied in the HTTP headers, along with a timestamp and version, as in the following example:

```
GET http://myaccount.blob.core.windows.net/mycontainer/myblob HTTP/1.1
Authorization: SharedKey myaccount:ctzMq410TV3wS7upTBcunJTDLEJwMAZuFPfr0mrrA08=
x-ms-version: 2009-09-19
x-ms-date: Sun, 27 Sep 2009 22:33:35 GMT
```

The base64 string in the header is not the actual access token, since that would render the API subject to replay attacks. Instead, this is a hash-based message authentication code (HMAC) using SHA256 constructed from the token and the message headers. The string to be signed when using the blob API is constructed as follows:

```
StringToSign = VERB + "\n" +
               Content-Encoding + "\n"
               Content-Language + "\n"
               Content-Length + "\n"
               Content-MD5 + "\n" +
               Content-Type + "\n" +
               Date + "\n" +
               If-Modified-Since + "\n"
               If-Match + "\n"
               If-None-Match + "\n"
               If-Unmodified-Since + "\n"
               Range + "\n"
               CanonicalizedHeaders +
               CanonicalizedResource;
```

The order of headers is fixed. If any of these headers are not part of the request, they can also be omitted from the string to sign by simply substituting a newline character. CanonicalizedHeaders is a string consisting of any headers beginning with the “x-ms-“ prefix, with the header names in lowercase, sorted alphabetically, with whitespace trimmed. Similarly, CanonicalizedResource is the storage resource targeted by the call, with any portions derived from the URI encoded exactly as used in the URI (i.e. the case and encoding in the signature must match that used in the URI.) The full details on the REST API authentication schemes and precisely how to construct a CanonicalizedResource string are found in MSDN at http://msdn.microsoft.com/en-us/library/dd179428.aspx

Storing a Blob

Storing a blob is similarly simple, using the HTTP PUT verb:

```
PUT http://accountname.blob.core.windows.net/containername/blobname
```

The request is required to include Authorization, date (x-ms-date), version (x-ms-version), type (x-ms-blob-type) and Content-Length headers to be accepted. An example block storage request could be as follows:

```
PUT http://myaccount.blob.core.windows.net/mycontainer/myblockblob HTTP/1.1
x-ms-version: 2009-09-19
x-ms-date: Sun, 27 Sep 2009 22:33:35 GMT
Content-Type: text/plain; charset=UTF-8
x-ms-blob-type: BlockBlob
x-ms-meta-item1: v1
x-ms-meta-item2: v2
Authorization: SharedKey myaccount:YhuFJjN4fAR8/AmBrqBz7MG2uFinQ4rkh4dscbj598g=
Content-Length: 11

Hello World
```

The x-ms-meta-*name* headers allow arbitrary metadata to be attached to the stored blob.

Reading from a Table

Though it is called Table Storage, after its primary use, tables are actually a collection of independent entities, each of is identified by a RowKey and has a bag of named properties. In addition, they can be spanned across multiple table stores using PartitionKeys for performance and load balancing.

In its simplest form, there are two ways to query entities from a table – getting an individual object by RowKey, or specifying a query string.

```
GET http://accountname.table.core.windows.net/tablename(PartitionKey='partitionkey',
RowKey='rowkey') HTTP/1.1

GET http://accountname.table.core.windows.net/tablename()?$filter=querystring HTTP/1.1
```

As with the Blob service, in the case of an authenticated table, the Authentication, x-ms-date, and x-ms-version headers must be provided, and a variety of other headers are available to specify how the data should be returned. An example of a table query by query string is as follows:

```
GET /myaccount/Customers()?$filter=(id%20ge%20100)%20and%20(ID%20le%20200) HTTP/1.1

Request Headers:
x-ms-version: 2009-09-19
x-ms-date: Thu, 01 Oct 2009 15:25:14 GMT
Authorization: SharedKeyLite myaccount: YhuFJjN4fAR8/AmBrqBz7MG2uFinQ4rkh4dscbj598g=
Accept: application/atom+xml,application/xml
Accept-Charset: UTF-8
```

This would request all entities in the table named Customers that have a property called ‘id’ with a value between 100 and 200. The service responds with an HTTP 200 containing an XML representation of the entities that match the query string (or the entity matching the specified RowKey and PartitionKey, if they were provided.) An example of a full response follows:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Wed, 23 Jun 2010 21:49:14 GMT
Content-Type: application/atom+xml;charset=utf-8
Server: Windows-Azure-Table/1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: a2f75534-32a6-47b5-bc1c-6775b09ca9fa
x-ms-version: 2009-09-19

741
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xml:base="http://myaccount.table.core.windows.net/"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">GuestBookEntry</title>
  <id>http://myaccount.table.core.windows.net/GuestBookEntry</id>
  <updated>2010-06-23T21:49:15Z</updated>
  <link rel="self" title="GuestBookEntry" href="GuestBookEntry" />
  <entry m:etag="W/&quot;datetime'2010-06-23T21%3A49%3A15.35328Z'&quot;">

<id>http://myaccount.table.core.windows.net/GuestBookEntry(PartitionKey='06232010',RowKey='12521249962450735715_
5c2dbee9-66d4-4e51-92ce-b8100055b635')</id>
    <title type="text"></title>
    <updated>2010-06-23T21:49:15Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="GuestBookEntry"
href="GuestBookEntry(PartitionKey='06232010',RowKey='12521249962450735715_5c2dbee9-66d4-4e51-92ce-
b8100055b635')" />
    <category term="myaccount.GuestBookEntry"
scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:PartitionKey>06232010</d:PartitionKey>
        <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
        <d:Timestamp m:type="Edm.DateTime">2010-06-23T21:49:15.35328Z</d:Timestamp>
        <d:GuestName>Test</d:GuestName>
        <d:Message>This is a message</d:Message>
        <d:PhotoUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:PhotoUrl>
        <d:ThumbnailUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:ThumbnailUrl>
      </m:properties>
    </content>
  </entry>
</feed>
0
```

The above example is a respoinse for a query that matched a single entity of type GuestBookEntry. The yellow text identifies the PartitionKey and RowKey of the matching entity, while the green text shows the entity's properties. These correspond to the primary key and columns in a traditional relational table.

The data returned by the table service will always present valid XML; if a property contains tags that would change the XML schema (e.g. angle brackets), they will be HTML encoded or enclosed in a CDATA tag as appropriate.

Writing to a Table

Writing to a table in Azure Table Storage consists of making an HTTP POST or PUT using XML data in the same format as that returned by a table GET. Thus, data to be written must be converted to XML text; binary data can be base64 encoded or enclosed in CDATA tags.

The previous example's entry could have been written by the following query:

```
POST http://myaccount.table.core.windows.net/GuestBookEntry HTTP/1.1
x-ms-version: 2009-09-19
x-ms-date: Wed, 23 Jun 2010 21:49:14 GMT
Authorization: SharedKeyLite myaccount:x/J5v2mHvkZhxF5pbFBvZTL6zBUOqIiRO1HvDaFK3Ws=
Accept: application/atom+xml,application/xml
Accept-Charset: UTF-8
Content-Type: application/atom+xml
Host: myaccount.table.core.windows.net
Content-Length: 1019

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
  <title />
  <updated>2010-06-23T21:49:14.9732353Z</updated>
  <author>
    <name />
  </author>
  <id />
  <content type="application/xml">
    <m:properties>
      <d:GuestName>Test</d:GuestName>
      <d:Message>This is a message</d:Message>
      <d:PartitionKey>06232010</d:PartitionKey>
      <d:PhotoUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:PhotoUrl>
      <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
      <d:ThumbnailUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:ThumbnailUrl>
      <d:Timestamp m:type="Edm.DateTime">0001-01-01T00:00:00</d:Timestamp>
    </m:properties>
  </content>
</entry>
```

Content-Type and Content-Length are required; as with any Azure storage API, authenticated requests must include the Authorization, x-ms-date, and x-ms-version headers. Note that the only schema referred to (in yellow) is the ADO Data Services schema on microsoft.com; no schema for the actual table (GuestBookEntry) is specified. The "columns" on this entity are defined only by the green section above, so it would be possible for each "row" in the nominal table to have different properties.

Each property must have a unique name; duplicate properties (e.g. providing two different <d:GuestName> tags) result in a 400 Bad Request error and the rejection of the entire POST.

Updates to an existing entity are similar, but are provided as an HTTP PUT rather than POST, and must include the PartitionKey and RowKey to be updated:

```
http://myaccount.table.core.windows.net/mytable(PartitionKey='myPartitionKey',RowKey='myRowKey1')
```

.NET API

The Windows Azure Platform is always interacted with using the REST API. However, to facilitate development using Azure Storage on Microsoft's premier development platform, the Microsoft .NET Framework 4.0, Microsoft has provided a Windows Azure SDK which contains managed interfaces to the Azure Platform. Since Windows Azure is most commonly used by developers who are familiar with Microsoft tools and platforms, these developers generally interact with Azure's Table Storage using Windows Communication Foundation classes, rather than directly interacting with REST calls.

The namespace Microsoft.WindowsAzure.StorageClient includes a variety of classes and interfaces that enable a .NET developer to wrap Azure Storage tables and blobs in familiar .NET classes. Using Microsoft.WindowsAzure.StorageClient, Azure tables can be interacted with using ADO.NET and LINQ queries.

A full description of the classes and methods in the Microsoft.WindowsAzure.StorageClient namespace can be found on MSDN at http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.storageclient.aspx

Table Storage using StorageClient

Using the Azure SDK for accessing tables involves some overhead and initial setup by the developer. The developer must create a class representing a single entity in the table (inheriting TableServiceEntity), with member properties corresponding to the properties the table entity will have in Azure Storage. In addition, a class representing the context must be created, inheriting TableServiceContext, which provides the appropriate credentials for access and implements IQueryable. Finally, the developer can call CloudTableClient.CreateTablesFromModel to create the tables in Azure Storage corresponding to the provided entity and context objects.

In return for this initial complexity, the Azure table then has an interface that inherits DataServiceContext, and can then be queried with LINQ or inserted into using ADO.NET, just like a SQL Server table. Since .NET will be enforcing the column names and types, this also makes Table Storage behave more like a traditional table, in that every entity must have the same properties. (Whether this is desirable or not depends on the application.)

It is important to understand that the SDK still converts queries to the REST API, and issues HTTP GETs, PUTs, and POSTs in response to the .NET calls. Thus, even the apparently-parameterized queries created in LINQ and ADO.NET are translated to text (URIs, HTTP messages, and XML ATOMs) before being sent to Azure Storage. StorageClient also performs encoding on data being stored before converting it into XML.

Blob Storage using StorageClient

Blob storage is interacted with using the CloudBlobContainer, CloudBlockBlob, CloudPageBlob objects.

A CloudBlobContainer represents a container, corresponding to a folder on Windows Azure Blob Storage. It contains a method, GetContainerReference, which takes the name of a container and implements methods to create and access blobs within. The CloudBlockBlob and CloudPageBlob objects represent individual blobs, and can be used to create, read, update, or delete them (including reads and updates to small portions of a larger blob.)

As with table storage, the SDK (via Windows Communication Foundation) converts queries to the REST API, and issues HTTP GETs, PUTs, and POSTs in response to the .NET calls. Container and blob names and properties are translated to text (URIs, HTTP messages, and XML ATOMs) before being sent to Azure Storage.

Amazon Simple Storage Service

Amazon's cloud platform, Amazon Web Services, offers a variety of storage options, including Simple Storage Service (S3), SimpleDB, and Relational Database Service (RDB.)

Simple Storage Service (S3)

Azure Blob Storage allows an application to store arbitrary blobs (binary objects), designated by a bucket name and an object name. One storage account can contain one or more buckets, each of which can contain any number of objects (to a limit of 5GB each) with associated attributes (metadata.) In addition, objects can be syndicated on Amazon's content delivery network, Amazon CloudFront.

A given bucket can either be public (accessed by anyone who knows the URL) or protected by signed URLs under a protocol called Signature Version 2. In the case of protected objects, access (including data definition or metadata queries) must come through API calls that provide an appropriate authentication header. Objects in public buckets can be accessed with a web browser.

SimpleDB

Amazon SimpleDB allows an application to store arbitrarily-formatted tabular data. Each row in SimpleDB is called an item, and may contain any number of named attributes. This can be used to represent a traditional database table, with the same attributes on each item in a domain, a single domain may contain items with different attributes. Thus, like Azure Table Storage, they are more akin to a set of entities each of which has a list of named properties than a table. A domain can be very large (billions of items.)

Data in SimpleDB is not relational. The database does not recognize a concept of foreign keys or enforce referential integrity. An application that requires these concepts will need to enforce them in application code, rather than relying on the underlying database to do it. SimpleDB can be accessed via REST or SOAP APIs, and Amazon provides SDKs for .NET and Java that wrap the REST API in a convenient class library. In addition, Amazon provides a Web Service Definition Language (WSDL) file for the SOAP API, so that developers can generate interfaces to SimpleDB.

Relational Database Service

An AWS developer who requires relational database functionality also has the option of using Amazon's Relational Database Service, which is a true relational database accessed with SQL rather than with REST or SOAP APIs.

RDS is a cloud-based implementation of the open-source MySQL database, with full support for that product's features. Access is via the MySQL Client/Server protocol on port TCP/3306, and any client library or toolset for MySQL can also interface with RDS.

From an application security perspective, RDS is identical to any other SQL-based database, and is thus out of scope for this paper.

REST API

Amazon Web Services uses a REST API for most S3 and SimpleDB requests.

Retrieving an Object

Accessing an existing object requires as issuing an HTTP 1.1 GET request:

```
GET accountname.s3.amazonaws.com/bucketname/key HTTP/1.1
```

For public objects, this is all that is required. Objects can also be restricted to their owner, another AWS user, or all AWS users; in this case, an authentication token must be supplied in the HTTP headers:

```
GET /bucketname/key HTTP/1.1
Host: accountname.s3.amazonaws.com
Date: Wed, 01 Mar  2010 12:50:00 GMT

Authorization: AWS 15B4D3461F177624206A:xQE0diMbLRepdf3YB+FIEXAMPLE=
```

The base64 string in the header is not the requester's actual AWS SecretAccess key, but rather the AWS Access Key ID (equivalent to a username) and a hash-based message authentication code (HMAC) using SHA1 constructed from the SecretAccess key and the message headers. The string to be signed when using the blob API is constructed as follows:

```
StringToSign = VERB + "\n" +
               Content-MD5 + "\n" +
               Content-Type + "\n" +
               Date + "\n" +
               CanonicalizedHeaders +
               CanonicalizedResource;
```

The order of headers is fixed. If any of these headers are not part of the request, they can also be omitted from the string to sign by simply substituting a newline character. CanonicalizedHeaders is a string consisting of any headers beginning with the "x-amz-" prefix, with the header names in lowercase, sorted alphabetically, with whitespace trimmed. Similarly, CanonicalizedResource is the storage resource targeted by the call, with the URI following some specific requirements. The full details on the REST API authentication schemes and precisely how to construct these strings is found in Amazon's documentation at http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAuthentication.html

Storing an Object

Storing a blob is done with the HTTP PUT or POST verbs:

```
PUT http://containername.s3.amazonaws.com/key HTTP/1.1
```

This request must include an AWS authentication header, storage class (x-amz-storage-class), and date (x-amz-date), and Content-Length headers to be accepted. An example block storage request could be as follows:

```
PUT http://grantbbh2.s3.amazonaws.com/obj1 HTTP/1.1
Content-Type: text/plain
Authorization: AWS AKIAI4GUZJBEQQBFEK3A:hHZpxb56pQ+i6XF6t6Rt4+9loms=
User-Agent: AWS SDK for .NET/1.0.9
x-amz-storage-class: STANDARD
x-amz-date: Mon, 05 Jul 2010 18:49:15 GMT
Host: grantbbh2.s3.amazonaws.com
Content-Length: 14

this is a test
```

x-amz-meta-*name* headers allow aribitrary metadata to be attached to the stored object.

Reading from a Table

SimpleDB tables are read with a simple REST query, and can even be read with a browser without specifying custom headers (though this would be impractical since some method to calculate the appropriate signature would be required.) A SimpleDB domain (table) is a collection of objects (rows), each of which is identified by a key.

To retrieve an object from the table, a URL can be sent like the following:

```
https://sdb.amazonaws.com/?Action=GetAttributes
&AWSAccessKeyId=accesskey
&DomainName=MyDomain
&ItemName=key
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&Timestamp=2010-07-10T15%3A03%3A07-07%3A00
&Version=2009-04-15
&Signature=signature
```

The key provided in the ItemName parameter (in yellow) indicates which object to retrieve. Since the access key and signature are provided in the URL, no custom headers are required. The API will respond with something akin to this:

```
<GetAttributesResponse>
  <GetAttributesResult>
    <Attribute><Name>Color</Name><Value>Blue</Value></Attribute>
    <Attribute><Name>Size</Name><Value>Med</Value></Attribute>
    <Attribute><Name>Price</Name><Value>0014.99</Value></Attribute>
    <Attribute><Name>Image</Name><Value encoding=”base64”>WW91IGNhb...wbGVEQ=</Value>
  </GetAttributesResult>
  <ResponseMetadata>
    <RequestId>b1e8f1f7-42e9-494c-ad09-2674e557526d</RequestId>
    <BoxUsage>0.0000219907</BoxUsage>
  </ResponseMetadata>
</GetAttributesResponse>
```

If a name or value contains invalid XML metacharacters (such as angle brackets, nulls, or undefined Unicode characters), the entire tag is Base64-encoded. CDATA tags or XML escape characters are not used.

To query a table based on the attributes (rather than pulling an entry by the key), a different request is used:

```
https://sdb.amazonaws.com/?Action=GetAttributes
&AWSAccessKeyId=accesskey
&DomainName=MyDomain
&SelectExpression=expression
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&Timestamp=2010-07-10T15%3A03%3A07-07%3A00
&Version=2009-04-15&Signature=signature
```

The select expression is a string provided in SimpleDB's SQL-like query language, which is fully documented in the AWS Developer Guide at http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/UsingSelect.htmlThe query string is a SQL-like string, and must be created by the developer.

Writing to a Table

SimpleDB tables are also written with a REST query. The put must specify which item is to be updated by a key:

```
https://sdb.amazonaws.com/
?Action=PutAttributes
&DomainName=MyDomain
&ItemName=key
&Attribute.1.Name=State
&Attribute.1.Value=fuzzy
&Attribute.1.Replace=true
&Attribute.2.Name=VersionNumber
&Attribute.2.Value=31
&Attribute.2.Replace=true
&Expected.1.Name=VersionNumber
&Expected.1.Value=30
&AWSAccessKeyId=accesskey
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&Timestamp=2010-01-25T15%3A03%3A05-07%3A00
&Version=2009-04-15
&Signature=signature
```

The PutAttributes action allows multiple attributes (columns) on the item (row) to be updated at once. The yellow section indicates which attributes are to be updated (State and VersionNumber in this example), what values are to be placed in the attributes, and whether this put should be allowed to overwrite existing values.

Table writes can be conditional. In this example, the green section lists expected values for an attribute (VersionNumber); if these values are not found on the item, the entire update fails (and is rejected with an error.) This field can be used for concurrency control, to prevent multiple updates from different applications or VMs from overwriting each other.

If a write succeeds, the response from the API just indicates the VM time used by the request:

```
<PutAttributesResponse>
  <ResponseMetadata>
    <RequestId>490206ce-8292-456c-a00f-61b335eb202b</RequestId>
    <BoxUsage>0.0000219907</BoxUsage>
  </ResponseMetadata>
</PutAttributesResponse>
```

SOAP API

In addition to the REST API, Amazon Web Services (including both S3 and SimpleDB) support a SOAP API, with a publically-available WSDL. The functionality of the SOAP API is identical to the REST API; only the syntax of the request is changed. The SimpleDB example above, in the SOAP API, would appear as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header
     xmlns:aws="http://security.amazonaws.com/doc/2007-01-01/">
      <aws:AWSAccessKeyId>accesskey</AWSAccessKeyId>
      <aws:Timestamp>2010-01-25T15:03:05Z</Timestamp>
      <aws:Signature>signature</Signature>
  </soapenv:Header>
  <soapenv:Body>
      <PutAttributesRequest xmlns='http://sdb.amazonaws.com/doc/2009-04-15'>
         <Attribute><Name>State</Name><Value>fuzzy</Value><Replace>true</Replace></Attribute>
        <Attribute><Name>VersionNumber</Name><Value>31</Value><Replace>true</Replace></Attribute>
         <Expected><Name>State</Name><Value>30</Value></Expected>
         <DomainName>MyDomain</DomainName>
         <ItemName>key</ItemName>
         <Version>2009-04-15</Version>
    </PutAttributesRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Traditional Database Attacks

In a web application, generally the database is not exposed to the Internet, and cannot be attacked directly. However, since the application receives input from users and then makes database calls based on that input, it is possible to attack the database indirectly, by using carefully-crafted malicious input. This is the foundation of almost all database attacks, both on traditional Relational Database Management Systems (RDBMS) and in the new world of cloud services.

SQL Injection

Structured Query Language (SQL) Injection is the best-known database attack, and is responsible for an enormous number of server compromises every year. SQL-based databases, such as Microsoft SQL Server, MySQL, or Oracle, use variants of the SQL language to issue commands to the database.

SQL Statements

SQL can be divided into two different kinds of commands:

- Data Definition Language (DDL): These SQL commands declare a database schema and instruct the server as to how to organize data. This includes verbs like CREATE, ALTER, and DELETE that create and modify tables and procedures, as well as commands that set permissions on objects.
- Data Manipulation Language (DML): These SQL commands add, modify, or delete items of data from the server. This includes the SELECT, INSERT, UPDATE, and DELETE verbs, which are the primary ways in which an application interacts with data in a database.

However, modern database servers often use special stored procedures to extend the functionalty of SQL. Executing these stored procedures can allow SQL code to carry out actions outside the scope of DDL and DML. For example, extended stored procedures could create entire databases, modify the accounts and permissions to the database server, cause the server to retrieve data from other sources via HTTP or other protocols, or even execute arbitrary code in the context of the database. The existence of these procedures means that a SQL injection attack does not just threaten data in the database – it runs the risk of compromising the entire service and providing the attacker with an entry point into the datacenter.

SQL Injection

The vulnerabilities that enable SQL Injection attacks all spring from the fact that SQL is a text string consisting of mixed code and data. For example, consider the following SQL statement:

```
SELECT ‘id’, ‘username’, ‘permissions’
FROM ‘users’ AS ‘u’ INNER JOIN ‘authorization’ AS ‘a’
WHERE ‘u.username’ == “grantb” AND ‘u.id’ == ‘a.id’;
```

This query is meant to retrieve the user ID and permissions of a user from a pair of tables, based on a username (presumably provided form a login form.) The username is *data*; it is an item to be looked up in the table. The remainder of the query is *code*; it instructs the database what to do. However, both are concatenated into a text string.

If the end-user had the ability to insert arbitrary text into their username, they could cause queries like these to be executed:

Cause the database to return data on every user in the system:

```
SELECT ‘id’, ‘username’, ‘permissions’
FROM ‘users’ AS ‘u’ INNER JOIN ‘authorization’ AS ‘a’
WHERE ‘u.username’ == “x” OR 1=1; --” AND ‘u.id’ == ‘a.id’;
```

Add a new user to the database with administrative permissions:

```
SELECT ‘id’, ‘username’, ‘permissions’
FROM ‘users’ AS ‘u’ INNER JOIN ‘authorization’ AS ‘a’
WHERE ‘u.username’ == “x” OR 1=1; INSERT INTO ‘users’ VALUES 1000, “hacker”; INSERT INTO PERMISSIONS VALUES
1000, “admin”; --” AND ‘u.id’ == ‘a.id’;
```

Use DDL for vandalism:

```
SELECT ‘id’, ‘username’, ‘permissions’
FROM ‘users’ AS ‘u’ INNER JOIN ‘authorization’ AS ‘a’
WHERE ‘u.username’ == “x”; DROP TABLE ‘users’; --” AND ‘u.id’ == ‘a.id’;
```

This is of course a very brief overview of SQL Injection; there is an enormous amount of research available on SQL Injection attacks and their mitigations. However, the mitigations all fall into three basic approaches:

- Input validation: in the examples above, the user is providing strings that contain quotation marks, dashes, semicolons, and other characters with a semantic meaning in SQL. If the user’s input is restricted to only characters reasonably found in the data item (e.g. letters and numbers only for a username), this makes attacks substantially more difficult.

- Encoding: if control characters are needed, an encoding function can be used to escape or transform any potentially problematic characters into a harmless form. URL Encoding is an example of one type of encoding used. Once again, this can make attacks substantially more difficult, but when data is parsed at multiple stages (e.g. by the web server, the application, and the database) coming up with a bulletproof encoding is challenging.
- Parameterization: Many programming languages, such as the .NET languages or Java, include data APIs that allow queries to be parameterized. In a parameterized query, the SQL query is sent with "placeholders" for the data elements, and then the data is sent separately. Since the database has been clearly told which elements are code and which are data, it is impossible for malicious input to change the functionality of the query. This is the preferred mitigation for SQL Injection attacks in most cases, as it is both easier to implement and harder to circumvent than input validation or encoding.

XML Injection

XML representations of data and XPATH representations of queries potentially suffer from the same issues as SQL, and for the same reason. An XML data item contains both the data itself and the schema for that data; an XPATH contains both the values to be queried for and the query itself. In both cases, code (in the sense of functionalty, not actual programming code) and data is being mixed.

When XML is used as a data storage mechanism, sometimes its structure can be manipulated by malicious data, just as is done in SQL. For the following examples, user records are stored as follows:

```
<user>
   <username>fred</username>
   <email>fred@contoso.com</email>
   <password>sdfyn2o49r</password>
   <uid>432</uid>
   <group>accounting</group>
   <hint>this is my password hint!</hint>
</user>
```

Assuming the application uses a standard DOM (Document-Object Model) parser, some values will break XML parsing in this structure. For instance, a username containing an angle bracket (<) will make the XML document invalid; likewise, a username containing the closing tag </username> will result in an additional closing tag with no corresponding opening tag. Comment tags <!-- and --> can also cause problems, and the & symbol (which defines an entity) will also cause a parser error.

However, it's possible to do rather worse than just causing parser errors. Note that in the above example, several fields are under control of the user – while user ID and group are probably assigned by the system, email address, password, and password hint are user input. Consider the following block of XML:

```
<user>
   <username>fred</username><!--</username>
   <email>fred@contoso.com</email>
   <password>sdfyn2o49r</password>
   <uid>432</uid>
   <group>accounting</group>
   <hint>--><email>fred@contoso.com</email><password>sdfyn2o49r</password><uid>0</uid>
<group>administrator</group><hint>this is my password hint!</hint>
</user>
```

In this example, our user has provided a username that includes the string `</username><!--` (shown in red), and a complex password hint (shown in yellow) containing a closing comment tag and a variety of XML. His email and password are valid, and his user ID and group have been inserted by the system as normal. However, when this record is read back by the system the resulting XML will be interpreted rather differently:

```
<user>
   <username>fred</username><!--</username>
   <email>fred@contoso.com</email>
   <password>sdfyn2o49r</password>
   <uid>432</uid>
   <group>accounting</group>
   <hint>--><email>fred@contoso.com</email><password>sdfyn2o49r</password><uid>0</uid>
<group>administrator</group><hint>this is my password hint!</hint>
</user>
```

Note that the username was terminated immediately by the closing tag that was actually part of the user's input. The actual email, password, user ID, and group tags are commented out by the opening comment tag from the username combined with the closing comment tag from the password hint. The password hint's XML contents then replace the actual tags with the attacker's own, changing his user ID to 0 and putting him in the "administrator" group. Finally, an opening password hint tag allows the XML to close successfully, such that the entire block is valid XML, yet gives the user elevated permissions.

Other XML interfaces use a SAX (Simple API for XML) parser rather than a DOM parser. These parsers are often not vulnerable to the same exploits as DOM parsers, primarily because many common parsers support a less-complete implementation of XML. Instead, they have other issues: a SAX parser reads XML serially, with later data superseding earlier data, and is generally much more tolerant of invalid or unclosed XML. As a result, the simplest attack on a SAX parser is to simply provide duplicate nodes. In our example above, an attack on a SAX parser might be to simply provide a password hint of `</hint><uid>0</uid><hint>` -- this closes the existing hint tag, adds a redundant user ID tag (thus superseding the previous one), and then adds a redundant hint tag again to ensure all tags are closed and the XML parses correctly.

An additional class of attacks against XML parsers is that of external entity injection. XML is extensible (hence the name) via defining new entity types within the document. If a user can inject code which closes the current XML tags, they can then follow it up by defining an entity that references something external, causing the parser to attempt to retrieve it. Example attacks include the following:

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE xy [
   <!ELEMENT xy ANY >
   <!ENTITY attack SYSTEM "file:///etc/passwd" >]><xy>&attack;</xy>
```

In this example, we begin a new XML document and declare a new document type. In our document type, we declare an entity type, called "attack", which contains the contents of a URI – in the case, the local UNIX password file. We then have the body of the document reference the “attack” entity. This is a fundamentally different sort of attack, since rather than trying to modify our data input to the database and cause it to have a different meaning on read, we’re actually forcing the server to take an action for us – in this case reading from the password file. Other actions we could take could include DoS against the parser by reading an endless file (e.g. /dev/random on a UNIX system) or making TCP scans behind the firewall (if I can craft XML that will return different errors depending on if the URL in the entity is resolved or not.)

Attacks on Cloud Storage Applications

Cloud storage services present a variety of possible avenues for attack, but as they use XML-based REST APIs, XML-based attacks may be fruitful. The purpose of this section is to walk through possible avenues of attack on a web site or service that is known to be using a cloud storage service such as Azure Storage or Amazon Web Servicesas its back-end.

Query String Injection

Amazon SimpleDB’s query language, though syntactically slightly different from SQL, string still follows the “SELECT *attributes* FROM *domain* WHERE *conditions* ORDER BY *sort_instructions* LIMIT *count*” format of a SQL query. If this is constructed from user input, unmodified SQL injection attacks may be possible. In SimpleDB the onus is entirely on the application developer to prevent SQL injection. Even when Amazon’s SDK is used, the entire SQL string (the “select expression”) is a single parameter, encoded as a unit, and subject to injection attacks. Likewise, in Azure Table Storage, the query string is a text string constructed by the application and passed to the API.

Amazon SimpleDB query strings support multiple quotation types (‘, “, `) and require escaping only for quotation marks not currently enclosed by another quotation type. This means that an attacker who can modify two different fields may be able to change the semantics of a query via injection:

```
select * from products where category = 'mycategory' intersection access = 'public' order by 'columnname' desc
```

The user-provided fields are in yellow. This can become:

```
select * from products where category = 'mycategory' or category = "' intersection access = 'public' order by '"
order by 'columnname' desc
```

When this is interpreted, the values are seen as:

```
select * from products where category = 'mycategory' or category = "' intersection access = 'public' order by '"
order by 'columnname' desc
```

The access-control portion of the query is thus stripped out by the presence of double quotation marks, and thrown into a value where it does no harm. The mitigation for this attack is identical to traditional SQL injection, however – validating user input before using it in a database query. This is made more difficult, however, by the absence of parameterized queries.

On the bright side, unlike SQL injection on a traditional SQL database, a SQL injection attack on either Amazon or Azure storage is capable only of changing the data returned by the query. While this can often be useful to an attacker (e.g. by causing the application to return another user's data, or to return a valid response to an invalid login attempt), it is not possible for an attacker to manipulate cloud storage query strings into changing the overall semantics of the request (e.g. turning a read into a write, or modifying or dropping entire tables.) Since the query string is nested into a REST query whose type determines the overall action (i.e. a PUT for database writes, GET for reads, etc.) and entirely different API calls are used for database configuration and DDL-type queries, there is no way for an attacker to use a malicious query string for anything but a database read. Database writes do not have a "query string" equivalent.

Direct Node Injection

However, it may be possible to attack the APIs that perform writes and other data operations. The actual parsers used by cloud storage engines like Windows Azure Storage and Amazon Simple Storage Service are not publicly revealed by their owners, so the nature of the parser is at first unknown. Thus, it's worthwhile to attempt a simple direct node injection, under the assumption that a SAX parser could be in use. Presented with a blog comment engine, one might be faced with fields such as comment title, contents, and user picture or avatar, while the system would be expected to add a timestamp and some sort of user identity (possibly taken from the session cookie of the logged-in user.) In Windows Azure Table Storage, it could be writing something like this:

```
  <content type="application/xml">
    <m:properties>
      <d:Title>This is a message</d:Title>
      <d:UserID>523</d:UserID>
      <d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl>
      <d:CommentText>This is a message</d:CommentText>
      <d:PartitionKey>06232010</d:PartitionKey>
      <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">2010-07-01T12:22:01</d:Timestamp>
    </m:properties>
  </content>
```

As an attacker, the order of the fields and their field names are "blind" – unless the application returns full error message information to the user, they have no way of knowing these except for raw guessing. This said, there have been many successful attacks carried out by blind SQL injection, so this should not be relied upon to protect the application.

A direct node injection might try providing a comment with the text of `</d:CommentText><d:UserID>100</d:UserID><d:CommentText>This is a message`. When inserted into the document, the result is:

```
<content type="application/xml">
    <m:properties>
      <d:Title>This is a message</d:Title>
      <d:UserID>523</d:UserID>
      <d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl>
      <d:CommentText></d:CommentText><d:UserID>100</d:UserID><d:CommentText>This is a message.</d:CommentText>
      <d:PartitionKey>06232010</d:PartitionKey>
      <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">2010-07-01T12:22:01</d:Timestamp>
    </m:properties>
  </content>
```

In this example, the cyan text is the UserID provided by the system. The red text is a d:CommentText tag which is closed by the malicious input, followed by the yellow d:UserID tag intended to replace the original tag with a new user ID. Finally, a new d:CommentText tag is opened so as to provide a comment to be posted under the other user's name.

On a naïve test application using Windows Azure Storage (i.e. an application that performs no defenses of its own against attacks from user input, passing everything directly to the underlying REST API), this attack is not successful. The presence of any duplicated tag results in the entire request being unceremoniously rejected with a reply of 400 Bad Request. This implies that Azure Storage is using a DOM parser and requires the entire XML document to validate in order to process it. Likewise, the Amazon SimpleDB SOAP API will reject requests with multiple `<Attribute>` tags for the same attribute.

CDATA and XML Comment Injection

However, the presence of a DOM parser implies that other avenues of attack may be more fruitful. If the parser is intelligent enough to reject out-of-hand any XML that contains duplicate nodes, it probably fully supports the XML format, including XML comments and the CDATA tag. Attempting XML comment injection, an attacker might try manipulating two different fields, in order to comment out intervening data. In this example, a title of `This is a message</d:Title><!--` is used, with a comment of `--><d:UserID>200</d:UserID> <d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-1b46389b855d.jpg</d:AvatarUrl><d:CommentText>This is a comment` When inserted into the document, the result is:

```
<content type="application/xml">
    <m:properties>
      <d:Title>This is a message</d:Title><!--</d:Title>
      <d:UserID>523</d:UserID>
      <d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl>
      <d:CommentText>--><d:UserID>200</d:UserID>
<d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl><d:CommentText>This is a comment</d:CommentText>
      <d:PartitionKey>06232010</d:PartitionKey>
      <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">2010-07-01T12:22:01</d:Timestamp>
    </m:properties>
  </content>
```

When this text is parsed, the green text is interpreted as an XML comment, starting in the Title tag and not ending until the CommentText tag. This excises the original UserID and AvatarURL tags from the request entirely, enabling the contents of the CommentText tag to inject new values for these fields.

This attack *does* work on a naïve test application using Windows Azure Storage. The Azure Storage API has full support for XML comments, and thus will parse the above input with the attacker's provided UserID and AvatarUrl, rather than using the ones intended by the application.

In addition to XML comments, the CDATA tag can be used to similar effect. CDATA is used to store data in an XML file that would otherwise conflict with the XML format, such as formatted text or HTML that may contain angle brackets, quotation marks, and other formatting characters. The CDATA tag is opened with the string `<![CDATA[` and closed with `]]>`. All characters between those sequences are treated as data, not as XML tags or entity references. The intent of this tag is to effectively escape any content such that it can be placed unaltered into XML, such as to embed an XML or HTML document into another XML document, but it can also be used by an attacker to elide tags just as the XML comment was above.

A title of `This is a message<![CDATA[` with a comment of `]]></d:Title><d:UserID>200</d:UserID><d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-1b46389b855d.jpg</d:AvatarUrl><d:CommentText>This is a comment` will result in:

```
<content type="application/xml">
    <m:properties>
      <d:Title>This is a message<![CDATA[</d:Title>
      <d:UserID>523</d:UserID>
      <d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl>
      <d:CommentText>]]></d:Title><d:UserID>200</d:UserID>
<d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl><d:CommentText>This is a comment</d:CommentText>
      <d:PartitionKey>06232010</d:PartitionKey>
      <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">2010-07-01T12:22:01</d:Timestamp>
    </m:properties>
  </content>
```

Note that this is very similar to, but not identical to, the XML comment attack shown above. The `</d:Title>` tag has been moved from the title string to the CommentText string, because to do otherwise would leave the CDATA tag "floating" between Title and UserID, not associated with any XML entity, resulting in a 400 Bad Request from the Azure Storage API. The disadvantage of this approach is that the Title tag now contains an assemblage of garbage data that will be echoed back to the user, but this may not be a problem in some applications. For instance, depending on the app, one may be able to inject the opening CDATA tag into a data field that is not displayed.

These same attacks are also usable on the Amazon SOAP API. The above request to Amazon SimpleDB might read as follows (with the malicious input for Title and CommentText highlighted):

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header
     xmlns:aws="http://security.amazonaws.com/doc/2007-01-01/">
      <aws:AWSAccessKeyId>accesskey</AWSAccessKeyId>
      <aws:Timestamp>2010-01-25T15:03:05Z</Timestamp>
      <aws:Signature>signature</Signature>
  </soapenv:Header>
  <soapenv:Body>
      <PutAttributesRequest xmlns='http://sdb.amazonaws.com/doc/2009-04-15'>
         <Attribute><Name>Title</Name><Value>This is a message<![CDATA[</Value></Attribute>
         <Attribute><Name>UserID</Name><Value>523</Value></Attribute>
         <Attribute><Name>AvatarURL</Name><Value>http://myaccount.blob.core.windows.net/
   guestbookpics/image_65fa49ae-1a73-4882-a15b-1b46389b855d.jpg</Value></Attribute>
   <Attribute><Name>CommentText</Name><Value>]]><Name>UserID</Name><Value>200</Value></Attribute>
         <Attribute><Name>AvatarURL</Name><Value>http://myaccount.blob.core.windows.net/
   guestbookpics/image_65fa49ae-1a73-4882-a15b-1b46389b855d.jpg</Value></Attribute>
   <Name>CommentText</Name><Value>This is a comment</Value></Attribute>
         <DomainName>MyDomain</DomainName>
         <ItemName>key</ItemName>
         <Version>2009-04-15</Version>
```

```
    </PutAttributesRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Persistent Cross-Site Scripting

In addition to attacks on the application via influencing underlying storage API calls, it is sometimes possible to perform either a persistent cross-site scripting (XSS) attack by leveraging the application's defense against injection attacks.

An application developer might attempt to mitigate injection attacks via HTML encoding strings before storing them in the database, or even place the entire string within a <![CDATA[block. However, due to the multiple layers of encoding and decoding involved, this method can easily introduce a cross-site scripting vulnerability.

The Windows Azure SDK includes a .NET assembly, Microsoft.WindowsAzure.StorageClient.dll, which performs exactly this sort of encoding. Since this assembly uses Windows Communication Foundation (WCF) Data Services, the encoding substantially mitigates injection attacks, but it should not be relied upon by itself to prevent cross-site scripting. Since the data services classes decode HTML when retrieving it from the database, the resultant markup will render when echoed back to the browser. Standard cross-site scripting mitigations, such as enabling ASP.NET ValidateRequest and performing input validation and output encoding on user-provided data should still be followed.

For example, using the blog-comment example above, an application using the StorageClient .NET API might receive a message or comment body containing HTML code intended for a cross-site scripting attack, such as `<script>alert('xss');</script>`. When this is passed into Azure Table Storage via StorageClient, the database will contain the following:

```
<content type="application/xml">
    <m:properties>
      <d:Title>&lt;script&gt;alert(&apos;xss&apos;);&lt;/script&gt;</d:Title>
      <d:UserID>523</d:UserID>
      <d:AvatarUrl>http://myaccount.blob.core.windows.net/guestbookpics/image_65fa49ae-1a73-4882-a15b-
1b46389b855d.jpg</d:AvatarUrl>
      <d:CommentText>&lt;script&gt;alert(&apos;xss&apos;);&lt;/script&gt;</d:CommentText>
      <d:PartitionKey>06232010</d:PartitionKey>
      <d:RowKey>12521249962450735715_5c2dbee9-66d4-4e51-92ce-b8100055b635</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">2010-07-01T12:22:01</d:Timestamp>
    </m:properties>
  </content>
```

While this protects from XML injection attacks, when StorageClient is queried for these values, the string returned will be the original one – `<script>alert('xss');</script>` -- and not the encoded one, and thus if it is echoed directly to the user a cross-site scripting exploit will occur.

This is not really an exploit of cloud storage, but rather a standard application-layer attack. However, it shows that validating and encoding user input can be harder than it sounds – sometimes data stored in the database is "user input" as well, and an application must be cautious with even its own stored data.

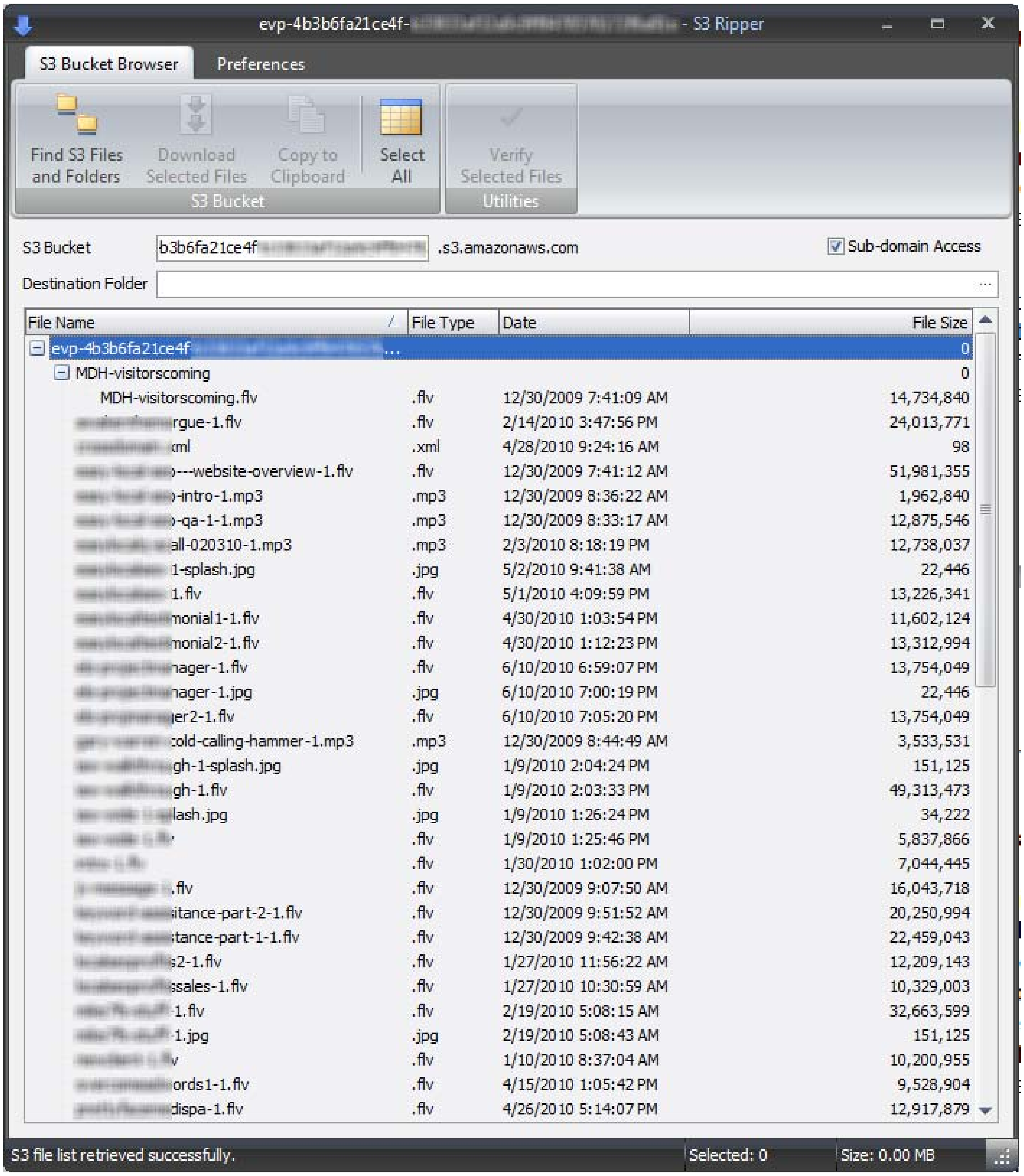
Enumeration

One common use of cloud storage, particularly blob storage, is storing files for end-user download. Since cloud storage systems offer very high capacity and bandwidth, along with affordable rates for data transfer, they appear ideal for this scenario. However, difficulties can ensue when cloud storage is used for files that either are not available to the general public, or are used in a multitenant application.

Many vendors sell downloadable content, such as ebooks, instructional videos, software, etc. They desire to make this content available to their end-users directly from the cloud, without also making it available for download by those who have not paid for the content. The usual method of protecting links is simple obscurity – that is, giving files and containers long, non-guessable names such as GUIDs. This is a particularly common practice for information products sold via affiliate marketing networks such as ClickBank.

However, as is always the case with security by obscurity, this is easily circumvented. In the simplest scenario, a paying customer can simply leak the URLs to his friends, enabling free downloads, or even post them on a public forum. However, in addition to this, an Azure container or Amazon bucket has an API allowing enumeration and other operations on the entire collection. Thus, armed with nothing more than the collection name (which is often found embedded in the source of public-facing web pages – after all, another common use of cloud storage is as an inexpensive CDN-like method of hosting static files like images, stylesheets, and videos), an attacker may be able to quickly discover *all* of the files offered for download. This can also be used to circumvent “tiered” access models, wherein someone can pay for a lower level access and be granted a higher level.

An easily-available product called S3 Ripper will, when given the name of an Amazon S3 bucket, quickly and easily enumerate all the files and folders it contains with their dates and filesizes, and enable a user to download its entire contents. The following page shows an example of S3 Ripper, provided only with the S3 bucket name of an information product (pulled from the source code of the product’s own promotional web site):



S3 Bucket b3b6fa21ce4f[redacted] .s3.amazonaws.com

Destination Folder

File Name	File Type	Date	File Size
⊟ evp-4b3b6fa21ce4f[redacted]...			0
⊟ MDH-visitorscoming			0
MDH-visitorscoming.flv	.flv	12/30/2009 7:41:09 AM	14,734,840
[redacted]rgue-1.flv	.flv	2/14/2010 3:47:56 PM	24,013,771
[redacted]xml	.xml	4/28/2010 9:24:16 AM	98
[redacted]---website-overview-1.flv	.flv	12/30/2009 7:41:12 AM	51,981,355
[redacted]-intro-1.mp3	.mp3	12/30/2009 8:36:22 AM	1,962,840
[redacted]-qa-1-1.mp3	.mp3	12/30/2009 8:33:17 AM	12,875,546
[redacted]all-020310-1.mp3	.mp3	2/3/2010 8:18:19 PM	12,738,037
[redacted]1-splash.jpg	.jpg	5/2/2010 9:41:38 AM	22,446
[redacted]1.flv	.flv	5/1/2010 4:09:59 PM	13,226,341
[redacted]monial1-1.flv	.flv	4/30/2010 1:03:54 PM	11,602,124
[redacted]monial2-1.flv	.flv	4/30/2010 1:12:23 PM	13,312,994
[redacted]nager-1.flv	.flv	6/10/2010 6:59:07 PM	13,754,049
[redacted]nager-1.jpg	.jpg	6/10/2010 7:00:19 PM	22,446
[redacted]er2-1.flv	.flv	6/10/2010 7:05:20 PM	13,754,049
[redacted]cold-calling-hammer-1.mp3	.mp3	12/30/2009 8:44:49 AM	3,533,531
[redacted]gh-1-splash.jpg	.jpg	1/9/2010 2:04:24 PM	151,125
[redacted]gh-1.flv	.flv	1/9/2010 2:03:33 PM	49,313,473
[redacted]lash.jpg	.jpg	1/9/2010 1:26:24 PM	34,222
[redacted]	.flv	1/9/2010 1:25:46 PM	5,837,866
[redacted]	.flv	1/30/2010 1:02:00 PM	7,044,445
[redacted].flv	.flv	12/30/2009 9:07:50 AM	16,043,718
[redacted]itance-part-2-1.flv	.flv	12/30/2009 9:51:52 AM	20,250,994
[redacted]tance-part-1-1.flv	.flv	12/30/2009 9:42:38 AM	22,459,043
[redacted]s2-1.flv	.flv	1/27/2010 11:56:22 AM	12,209,143
[redacted]ssales-1.flv	.flv	1/27/2010 10:30:59 AM	10,329,003
[redacted]1.flv	.flv	2/19/2010 5:08:15 AM	32,663,599
[redacted]1.jpg	.jpg	2/19/2010 5:08:43 AM	151,125
[redacted]v	.flv	1/10/2010 8:37:04 AM	10,200,955
[redacted]ords1-1.flv	.flv	4/15/2010 1:05:42 PM	9,528,904
[redacted]dispa-1.flv	.flv	4/26/2010 5:14:07 PM	12,917,879

S3 file list retrieved successfully. Selected: 0 Size: 0.00 MB

The availability of this and similar tools make direct exposure of non-public files in public buckets inappropriate. In addition, this issue can be a problem in multitenancy situations – if a web application is hosting files for more than one customer, even if the files are public, the application's full customer list may not be intended to be disclosed. However, assuming public buckets are used, any one customer could determine the other customers simply by querying the cloud storage back-end.

Similar tools for Windows Azure Storage do not currently exist. The usefulness of such a tool would be limited, since when a container is set to public in Windows Azure, the developer has the option to specify if public access applies only to the blobs themselves, or to the container and its associated metadata.

If the developer properly sets the container ACL to allow public access to blobs only (via setting `x-ms-blob-public-access` to `blob` rather than `container`), then enumeration is not possible, and will return only:

```
<Error>
   <Code>ResourceNotFound</Code>
      <Message>The specified resource does not exist.
               RequestId:65f11e15-378c-4456-bd23-a21650c6cad3
               Time:2010-07-04T18:51:56.7048688Z</Message>
</Error>
```

Note that the error message for a container that does not have public access to container data is identical to one for a container that does not exist at all, so these requests cannot even be used to attempt to identify container names via brute force or process of elimination.

If public access is set to the container level, the container supports the List Blobs command with no authentication requirement – as a result, a container can be enumerated using a standard web browser. For instance:

```
GET http://grantbbh.blob.core.windows.net/guestbookpics?restype=container&comp=list HTTP/1.1
```

will return the following:

```
<?xml version="1.0" encoding="utf-8"?>
<EnumerationResults ContainerName="http://grantbbh.blob.core.windows.net/guestbookpics">
   <Blobs>
      <Blob>
         <Name>image_0b395f75-0ced-45e4-9cea-af51a9683f0f.jpg</Name>
         <Url>http://grantbbh.blob.core.windows.net/guestbookpics/image_0b395f75-0ced-45e4-9cea-
af51a9683f0f.jpg</Url>
         <Properties>
            <Last-Modified>Tue, 15 Jun 2010 20:38:23 GMT</Last-Modified>
            <Etag>0x8CCDAE1D54F42CF</Etag>
            <Content-Length>620888</Content-Length>
            <Content-Type>image/pjpeg</Content-Type>
            <Content-Encoding />
            <Content-Language />
            <Content-MD5 />
            <Cache-Control />
            <BlobType>BlockBlob</BlobType>
            <LeaseStatus>unlocked</LeaseStatus>
         </Properties>
      </Blob>
      <Blob>
         <Name>image_0b395f75-0ced-45e4-9cea-af51a9683f0f_thumb.jpg</Name>
         <Url>http://grantbbh.blob.core.windows.net/guestbookpics/image_0b395f75-0ced-45e4-9cea-
af51a9683f0f_thumb.jpg</Url>
         <Properties>
            <Last-Modified>Tue, 15 Jun 2010 20:38:26 GMT</Last-Modified>
            <Etag>0x8CCDAE1D706C20F</Etag>
            <Content-Length>4285</Content-Length>
            <Content-Type>application/octet-stream</Content-Type>
            <Content-Encoding />
            <Content-Language />
            <Content-MD5 />
            <Cache-Control />
            <BlobType>BlockBlob</BlobType>
            <LeaseStatus>unlocked</LeaseStatus>
         </Properties>
      </Blob>
   </Blobs>
[...]
<NextMarker />
</EnumerationResults>
```

This contains the same information used by S3 Ripper. While current Windows Azure tools, such as the Windows Azure MMC (http://code.msdn.microsoft.com/windowsazuremmc) and Azure Storage Explorer (http://azurestorageexplorer.codeplex.com) require the SubscriptionID and a valid management certificate to enumerate contents of blob containers, this is only because a preliminary step (enumerating which containers belong to an account) requires this. While setting the container public at the container level is only appropriate for applications where enumeration and the ability of the public to perform actions other than reading on the container is desirable, and thus not for most situations, some application developers may mistakenly do this due to not understanding the distinction. SDK sample applications generally set containers public at the container level.

More secure ways to distribute public files from cloud storage is discussed in the mitigations section below.

Account Attacks

In addition to writing secure applications, application developers must also follow reasonable account security procedures. Both Windows Azure and Amazon Web Services allow an administrative user to authenticate using only an email address and password. If this avenue is open to an attacker, there is no need to attack an application – he can simply download the entire contents of the account.

The email address and password for the administrative owner of an AWS or Azure account should be treated like any other administrative credential, like a Domain Admin or root password. The administrator should never use the same email address and password as is used for their standard Amazon shopping account or for other Windows Live services like Messenger. For the email address, ensure an email address fully controlled by your organization is used – never use an email address from a public webmail provider, as these often have “password reset” functions based on easily-guessed cognitive passwords (e.g. your mother’s maiden name, what school you attended, etc.) Entire enterprises have been compromised before by simply resetting the administrator’s password and guessing the password reset questions.

In addition, both Amazon and Microsoft offer multifactor authentication options, which should be utilized whenever possible. These are discussed in the mitigation section below.

Defenses and Mitigations

Standard Mitigations

Since most of the attacks on cloud storage are simply modern, updated variants of injection attacks and persistent cross-site scripting, some of the most effective mitigations are already well understood. Following good software development practices such as the Security Development Lifecycle, as well as the security recommendations from whitepapers such as Microsoft's Security Best Practices for Developing Windows Azure Applications (coauthored by the author of this paper) and Amazon's Security Best Practices 2010 will go a long way toward protecting applications in the cloud.

Input Validation

Since these exploits rely on malicious input from users, input validation is vital. Platform-layer mitigations like ASP.NET's ValidateRequest (enabled by default in current versions of Visual Studio) help by filtering out the most straightforward attacks, as does using the Microsoft.WindowsAzure.StorageClient classes when accessing the Azure API. However, as these are a blacklist-based and encoding-based mitigation, respectively, neither is a substitute for whitelist-based input validation. When user input is being processed, any characters that do not match the expected data type should be rejected.

Input validation is easy for fields with obvious data types, like a ZIP code or a SHA-256 hash. However, more care must be taken to validate input when an application expects data such as XML or HTML from a user (especially when non-English Unicode characters are accepted for internationalization purposes.)

For defense-in-depth, input validation should be used on all user input (not just obvious input such as HTML form fields, but also web service calls, URL parameters, and values from cookies), while platform mitigations such as ValidateRequest (on ASP.NET) and mod_security (on Apache web servers) are enabled.

Encoding

Encoding attempts to prevent attack by ensuring that data always remains data, and is never misinterpreted as code. When user data is to be put into cloud storage and it is not strongly-typed data that has been verified against a whitelist, that data should be encoded. Since the APIs are XML, it is also possible to wrap the possibly-malicious data in a `<![CDATA[` block, though care must still be taken to avoid malicious input containing the CDATA closing sequence `]]>`.

XML encoding is fairly standard; there are only a few control characters that need to be encoded (angle brackets, single and double quotes, and ampersands.)

Encoding is also used to mitigate cross-site scripting, by encoding output before sending it to the user's browser. While this is an effective mitigation, be aware that encoding for data storage (which is reversed upon reading from the storage) and encoding to mitigate cross-site scripting are separate mitigations – one does not obviate the need for the other. Multiple encoding attacks can be highly sophisticated, relying on some layers of encoding and decoding to attack other layers (e.g. sending input that is encoded already, such that it passes through the encoder unchanged and is turned into malicious output by the application's own decoder), and defending against them is beyond the scope of this paper.

Windows Communication Foundation

For Windows Azure development, most developers are protected from injection attacks against Table Storage due to the way they access the underlying database. Rather than making REST calls directly, authors of ASP.NET applications can users wrap Azure Table Storage in WCF Data Services classes (such as the previously-mentioned StorageClient class), which perform basic encoding is automatically. As a result, injecting XML into the underlying REST calls should be impossible, since application logic is no longer involved in constructing the REST call – it is taken care of by the WCF class libraries, which properly escape XML metacharacters. This is roughly akin to making parameterized queries to avoid SQL injection – the developer does not construct queries from strings, but rather allows the platform to make the query based on the parameters provided.

No vulnerabilities in this encoder were found during the research of this paper. However, due to the complexity of encoding attacks, defense-in-depth practices should still be followed rather than relying entirely on the platform for protection.

In addition, applications consuming Windows Azure Storage from a PHP application (or other non-.NET based platform) must still mitigate these issues at the application layer, as must applications using other cloud providers like Amazon Web Services. This mitigation is not a function of the Azure platform itself, but rather of Windows Communication Foundation and ASP.NET. Likewise, developers using Amazon SimpleDB do not necessarily have this option available to them, unless they happen to be using the ASP.NET platform.

SOAP Toolkits

When using Amazon's SOAP API, developers can be subject not only to the query string injection possibilities, but XML injection as well. However, as with WCF developers on Azure, most developers using Amazon's SOAP API are protected against injection attacks due to the way they access the database.

Developers using the REST API often do so directly, writing code that assembles the REST URLs and interprets the response. However, SOAP users generally use some sort of SOAP toolkit to interpret the WSDL and generate interface classes. These toolkits may include the System.Web.Services classes in the .NET Framework for Windows developers, or similar SOAP classes available for Java, PHP, and other platforms. The interface code generated by these toolkits is often designed to contain harmful content (such as XML metacharacters, nulls, or undefined Unicode characters) inside CDATA blocks, encoding, or escaping.

As with WCF, no vulnerabilities in these encoder were found during the research of this paper. However, due to the complexity of encoding attacks, defense-in-depth practices should still be followed rather than relying entirely on the platform for protection.

In addition, applications consuming Amazon Web Services using the REST API must mitigate injection issues at the application layer. Amazon Web Services does not require the use of any particular SOAP toolkit (indeed, developers have the option of creating SOAP queries programmatically from strings, it would just be unusual to do so) so this mitigation is not a function of the cloud services provider, but rather of the development platform being used.

Encryption and Multifactor Authentication

As well as good application development, owners of Windows Azure and Amazon Web Services applications should take care not to allow attackers even easier avenues of attack on their services. Use HTTPS for all communication with storage that passes outside the Amazon or Azure perimeters; indeed, there is little reason not to use HTTPS always, even for communication coming from Amazon EC2 or Azure Hosted Services.

Use a unique email address and password that are not used for any other purpose. Ensure that address belongs to your organization and not to a public email provider that has “password reset” functionality. Failing to properly protect the administrative credential can give attackers a back door into the entire storage infrastructure.

Finally, Amazon Web Services offers multi-factor authentication, wherein a dynamic password token is required to log in as well as the username and password. Windows Azure offers the ability to perform all Azure management functions using certificates, and these certificates may reside within a TPM or smart card. (Note, however, that in Azure the administrator’s Live ID and password still grants access; however, once certificates are established there is no reason for this credential to be used on a day-to-day basis so knowledge of it can be kept tightly controlled.)

Ineffective Mitigations

Enumeration attacks, in particular, have been subject to many ineffective or counterproductive attempts at mitigation, generally by companies selling a security product. All of these products aim to prevent a single paid user from giving away a site’s contents via simply posting the cloud storage URL onto a public forum. In order to hide the bucket name and prevent enumerating filenames, these products have attempted a varity of approaches:

- Providing links only to a Flash application on a standard webhost (or EC2 instance) which contains the S3 link and performs the download. This application is then normally placed on a script that performs referrer checking.
- Registering a separate domain name that is CNAMEd to the bucket at *bucketname*.s3.amazonaws.com.
- Providing links to a script on a standard webhost, behind an access-controlled gateway, which then performs the download and returns the results in the HTTP response.

The Flash application approach is easily bypassed by running a debugging proxy (such as Fiddler or Burp) and monitoring the Flash application’s web requests, which still contain the cloud hosting URL. Likewise, using a Flash debugger can extract the URLs from within the application.

The CNAME approach is similarly flawed since a user can simply use common DNS tools such as dig to find out the contents of the CNAME. This only protects against an attacker who is unaware of any tools outside of a web browser.

Using a script to perform the download and return the result, essentially acting as a proxy, is effective for controlling access. However, it entirely defeats the purpose of cloud storage – since the data stream is entirely proxied through the standard webhost, any performance gains from cloud storage are negated and the bandwidth cost is actually multiplied (since one must both pay the cloud storage provider for bandwidth and then pay the standard webhost *twice*, once for the fetch and once for the return.)

All of these methods are attempts to avoid the complexity of using Windows Azure's Shared Access Signatures or Amazon S3's Signed URLs. In S3, signed URLs are the only effective way to protect access without using the counterproductive proxy model. Under Windows Azure, the correct mitigation is to set public access at the blob level rather than the container level, making enumeration impossible.

Application Design Mitigations

Many of the issues discussed so far have been simple code bugs – i.e. mistakes made by developers that allow an exploit to occur even in a properly-designed application. However, some of these issues can also be mitigated via proper application design, which can also provide defense in depth against other types of attacks.

Shared Access Signatures

Since hiding URLs for files stored in cloud storage does not provide protection, how then should an application that wants to make use of cloud storage's cheap capacity and bandwidth also make files available only to paying customers?

Both Windows Azure Storage and Amazon S3 provide methods for access-controlled storage blobs, called Shared Access Signatures in Azure and Signed URLs in S3. These enable placing an access token directly in a browser-accessible URL that will enable the legitimate user to download the file without giving them the ability to share it with others in perpetuity.

An Azure Shared Access Signature consists of a normal Azure blob storage REST URL, followed by fields indicating the access policy, signed resource, optional signed identifier, and digital signature that both specify the access to be allowed and verify that this specification is authorzed. A signature can apply either to a specific blob or to an entire container.

Instead of making a public container to store members-only files, the container is instead created with access control enabled. Then, rather than providing the user with a direct link to the file in the access-controlled area of the website, the user is given a link to an ASP.NET page that generates a URL to the access-controlled blob container, with an associated expiration time, and then redirects the user's browser to that URL. The appropriate expiration time depends on the size of the file; for reasonably small files, an hour is probably sufficient. In addition,if there are different membership tiers, they can be placed in separate blob containers, so that the signature for one is not usable for the other.

For example, a website has downloadable music in a blob container as follows:

```
http://myaccount.blob.core.windows.net/musicdownloads
```

Rather than making this container public and attempting to keep the links hidden (which would generally entail giving the account and container very long, GUID-based names), they instead mark it private and never give this URL to end-users (who could not use it anyway.) They create an ASP.NET script (e.g. download.aspx?file=filename.mp3) which then generates a link such as the following:

```
http://myaccount.blob.core.windows.net/musicdownloads/filename.mp3?st=2010-07-28T11:20Z&se=2010-07-
29T12:20Z&sr=c&sp=r&si=YWJjZGVmZw%3d%3d&sig=dD80ihBh5jfNpymO5Hg1IdiJIEvHcJpCMiCMnN%2fRnbI%3d
```

This is a standard URL; no additional headers or API calls are required for the user to use it. The same script that generates the URL then sends a redirect to the user's browser to go to it, creating a seamless user experience.

The portions of the URL are as follows:

```
st=2010-07-28T11:20Z
se=2010-07-09T12:20Z
```

These indicate the starting and ending time of the signature's validity. Outside of that time range, Azure Storage will not consider the links valid. Any ISO 8061 date and time can be used, and UTC time is assumed if no time zone specifier is issued.

```
sr=c
```

This indicates that the signed resource is the container – that is, these URL parameters are applicable to all blobs in the container. Alternately, `sr=b` would indicate it applies to the blob alone.

```
sp=r
```

This indicates that only read permissions are allowed; write, delete, and list are also available if specified.

```
si=YWJjZGVmZw%3d%3d
```

This is a signed identifier – a reference to a container-level access policy. While this field is optional, it also provides the option of revocation; by changing the identifier of the container (which does not require renaming or moving it), all previously-issued signed URLs can be rendered invalid.

```
sig=dD80ihBh5jfNpymO5Hg1IdiJIEvHcJpCMiCMnN%2fRnbI%3d
```

Finally, this signature ensures that the previous strings have not been tampered with. It consists of an HMAC-SHA256 generated based on a concatenation of the above fields, signed using the Azure account's API key (which is not available to end-users.) This end-user could download this file or any other file in the container for the next hour (due to the start and end times specified), but posting the URL on a public forum would be of limited use, since the URL will become invalid an hour later. Meanwhile, a legitimate user would be totally unaffected, since they will be typing in only the friendly, access-controlled download.aspx URL and not the Azure URL at all, yet they will download directly from cloud storage.

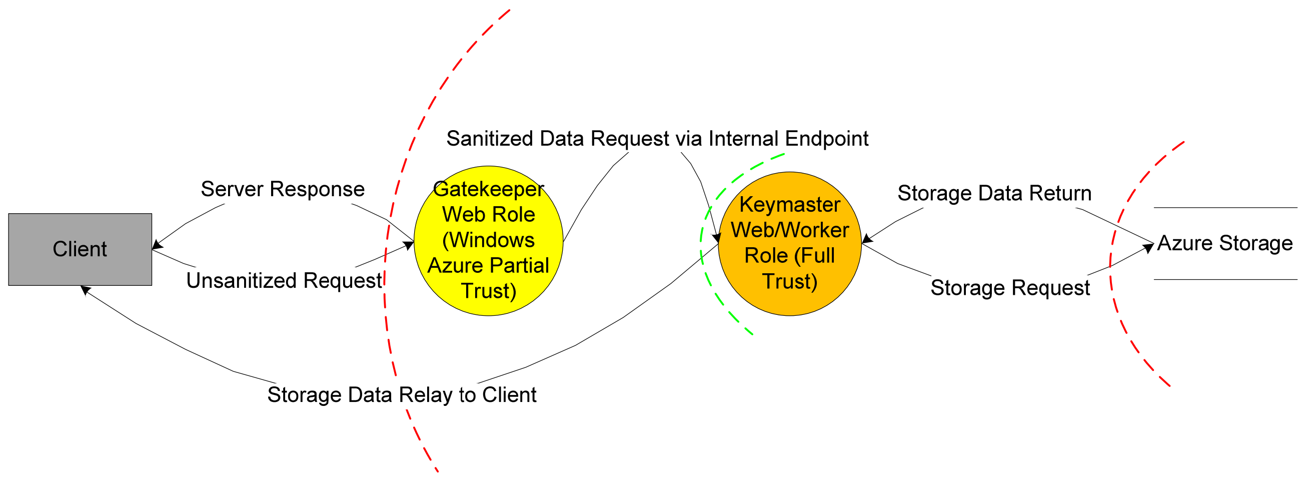
Amazon S3 signed URLs work identically, with the ability to specify validity times and blob- or bucket-level permissions, but with a different syntax.

Gatekeeper Design Pattern

In the standard datacenter service world, defense in depth is often achieved with 2-tier or 3-tier service architectures, where each tier has only very limited access to the lower tiers (generally through a router or firewall.) This helps limit the opportunity for cascading compromise, and ensures that an attacker who compromises a service gets only the bare minimum of access and still has a difficult task ahead of him if he wishes to acquire sensitive data.

A similar method can be used in cloud services, by following what is termed a gatekeeper design pattern. Windows Azure facilitates this with the Windows Azure Partial Trust feature, which enables a web role to be run with sharply limited privileges.

A gatekeeper design pattern uses brokered access to storage, which limits the attack surface on sensitive data via using intermediate, privileged roles. These roles are deployed on separate VMs, so that breaking into one web role does not give special access onto the other.


Sanitized Data Request via Internal Endpoint
Server Response
Gatekeeper Web Role (Windows Azure Partial Trust)
Keymaster Web/Worker Role (Full Trust)
Storage Data Return
Client
Azure Storage
Unsanitized Request
Storage Request
Storage Data Relay to Client

In this example, the Gatekeeper is a web role which services requests from the Internet. This role manages only presentation-layer logic – it validates the input it receives, and displays web sites to Internet users. This role can run entirely under Windows Azure Partial Trust, such that it is difficult to escalate an application-layer compromise. It does not contain any storage credentials, such as SQL Azure logins or Azure Storage shared keys – it only contains enough information to authenticate itself to the Keymaster role.

The Keymaster role runs on a separate VM instance, and accepts authenticated input *only* from the Gatekeeper (using WCF signatures, HTTPS mutual authentication, or some other method.) End users have no way to connect to it and are unaware of its existence. If the application for some reason requires full-trust operations (e.g. to run a legacy, native DLL), the Keymaster can be set to run as Full Trust, but ideally it runs under Windows Azure Partial Trust as well. This role contains the storage credentials for the application's SQL Azure or Azure Storage, and brokers all storage requests for the Gatekeeper.

The Keymaster must still perform some input validation in case the Gatekeeper has been compromised, but since it has a very limited interface (exposing methods only for the specific operations the Gatekeeper is expected to perform), it is easier to secure. Should an attacker compromise the Gatekeeper role, they have no way to perform arbitrary queries against data; they're limited only to the types of actions that would be expected. (For example, on an ecommerce site, they may have interfaces only to get one user's data at a time, since there is no reason for an ecommerce site's customer-facing application to be able to retrieve data on many users at once.) Of course, using this design pattern requires being intelligent about the interfaces the Keymaster exposes – having an interface for "execute arbitrary SQL," for instance, defeats the purpose of this separation.

This same design pattern can be used within Amazon's Elastic Compute service, by setting up multiple VMs each running a different part of the application. While Amazon EC2 is an infrastructure-as-a-service (IaaS) product and does not have a feature comparable to Windows Azure Partial Trust, the machine playing the Gatekeeper role can still be hardened against attack and provide the same separation of privileges and defense in depth capabilities.

Conclusion

Cloud storage systems like those offered by Microsoft Windows Azure and Amazon Web Services offer a great deal of promise, but just as with traditional data storage methods, they also offer opportunity for attackers to take advantage of careless developers. Developers must ensure that they follow defense-in-depth and least-privilege practices, taking advantage of cloud storage features to protect data when possible, yet also continuing best practices like input validation and proper encoding.

Windows Azure Storage applications will likely tend to be less vulnerable to database attacks than traditional SQL-based applications, simply because access to Azure Storage is almost always from ASP.NET applications, which are largely protected by platform mitigations such as request validation and the encoding performed by WCF Data Services. However, any application directly issuing REST queries to any cloud storage provider may still introduce vulnerabilities at the database layer.

Appendix A: Additional Resources

Security Best Practices for Windows Azure Applications
http://download.microsoft.com/download/7/3/E/73E4EE93-559F-4D0F-A6FC-7FEC5F1542D1/SecurityBestPracticesWindowsAzureApps.docx

Microsoft Security Development Lifecycle (SDL) Portal
http://www.microsoft.com/sdl

Windows Azure Portal
http://www.microsoft.com/windowsazure/windowsazure/

Windows Azure Developer Portal
http://dev.windowsazure.com

Amazon Web Services
http://aws.amazon.com

Microsoft’s Compliance Framework for Online Services
http://www.globalfoundationservices.com/documents/MicrosoftComplianceFramework1009.pdf

Securing Microsoft's Cloud Infrastructure
http://www.globalfoundationservices.com/security/index.html

About Cross-Frame Scripting and Security
http://msdn.microsoft.com/en-us/library/ms533028(VS.85).aspx

Appendix B: Glossary

Developer Portal	The Windows Azure Developer Portal is an administrative portal for managing, deploying and monitoring Windows Azure services. The Developer Portal can be accessed at http://windows.azure.com
Fabric	The logical clusters of machines which provide a role execution environment inside a virtual machine in Windows Azure
Partial Trust	Partial trust is a concept in .NET that allows executable code to run with reduced capabilities such as the ability to print, make a socket connection or open files.
REST	REpresentational State Transfer; a software design that uses a stateless client-server architecture in which the web services are viewed as resources and can be identified by their URLs.
SAML	Security Assertion Markup Language. An XML-based industry standard for exchanging authentication and authorization information. Frequently used in SOAP APIs.
SDL	The Microsoft SDL is a security assurance process that is focused on software development. It is a collection of mandatory security activities, grouped by the phases of a software development life cycle (SDLC). You can learn more about the Microsoft SDL at http://www.microsoft.com/security/sdl
VM	A software emulation of a computer that runs in an isolated partition of a real computer. Windows Azure Web Roles and Worker Roles, and Amazon EC2 Instances run on VMs.
WCF	Windows Communication Foundation, a series of .NET classes used for network communication. Includes WCF Data Services, which wraps underlying databases in an abstraction layer.
Web Role	A web role is a role in Windows Azure that is customized for web application programming as supported by IIS 7 and ASP.NET.
Worker Role	A worker role is a role in Windows Azure that is useful for generalized development and may perform background processing for a web role.