Hardening your Windows 8 apps for the Windows Store

Bill Sempf

The Microsoft security story has changed. Windows 7 was so secure that malware authors started turning their attention to the simpler and less robust domain of mobile platforms like Android and iOS.

With Microsoft entering the foray (again) however, there are going to be some opportunities to attack devices through Windows Store apps. Here we consider the platform for Windows 8 app development, how testing is usually performed, and the features and security countermeasures provided by WinRT. We conclude with a set of 'good ideas' that will hopefully keep everyone off the front page of the local paper.

An overview of development for Windows 8 Store App

Windows Store apps are very different from Windows Desktop apps. Windows Store apps follow in the footsteps of iOS and Android development, providing a relatively small instruction set, a deployment and lifecycle model, and a sandbox for execution.

Additionally, Microsoft developed a totally new Windows SDK for the development of Windows Store apps: WinRT. WinRT has development hooks in HTML5, .NET and C++.

HTML5

For the first time, HTML is a first-class citizen in the Windows development world. Using JavaScript, Cascading Style Sheets and some Microsoft magic-glue goodness, it is possible to build a Windows Store app just like one would build a web site – JQuery, image tags and all.

Of course, this brings with it some classic security considerations. HTML Windows Store apps are rendered using the IE10 rendering engine and are subject to some of the same vulnerabilities. Windows Store apps that are built using HTML5 are XSSable, injectable, and valid for nearly any of the normal tricks you use for attacking web applications. Additionally, as they usually get their data from a backend service via REST, there is a service layer to consider as well.

Windows JavaScript

The Windows development group has embraced JavaScript. The JavaScript language isn't very well suited for Windows development, as it lives on the user thread, and is loosely typed to the extreme. With the addition of the WinRT API, and some additional API work by the JavaScript team, it makes a more than acceptable hacking language for even complex apps.

It should be noted that pretty much any library for JavaScript development works in Windows Store apps built with HTML5 as well. JQuery, knockhou.js, and node are just a few of the things I have seen people use. That is a two sided sword, though of course. While it makes development even easier, it also means any security flaws found in those products also reside in any app that uses them.

The WinJS library

To further ease transition from web to app development for JavaScript programmers, Microsoft produced the WinJS library. WinJS gives app developers a lot of what they expect in the environment, which JavaScript doesn't already provide. For instance:

- Asynchronous behavior through Promises
- Navigation support
- Classes and Namespaces
- Touch friendly UI components

XAML

Those who have done development in Silverlight or WPF or Windows Phone will remember XAML. XAML is the other first class user interface markup language in the Windows universe, and it has a considerably lower attack profile than HTML5. XAML is rendered using COM and has much less flexibility than HTML.

C#, VB and the.NET stack

At the developer level, the XAML world is all happy .NET: VB.NET, C# and C++. Experienced .NET developer will notice that the .NET library for Windows Store apps is significantly smaller than what is found in the desktop version. Anything that is not allowed by the Windows Store sandbox rules has been removed, and some of the omissions are startling.

The .NET For Windows Store Apps (as it is called) includes:

- Most of System.Collections
- Enough of System.ComponentModel to provide for DataAnnotations
- System.Diagnostics
- System.Dynamic
- System.Globalization
- Just the stream compression from System.IO
- Only HTTP stuff form System.Net
- A crippled System.Reflection
- System.Runtime
- The identity stuff from System.Security
- System.ServiceModel
- Regex from System.Text
- System.Threading
- System.Xml
- XAML controls from System.UI
- The DLR in Microsoft.CSharp
- Compilation tools from Microsoft.VisualBasic

C++ and DirectX

The last major platform for Windows Store development is C++ and DirectX. This differs from using C++ and the .NET stack, as C++ .NET uses XAML as markup.

DirectX is the bare-bones graphics library for Windows, and has its own set of problems. For instance, when DirectShow first came out in 2009, it included a remote code execution vulnerability for QuickTime videos. Anytime you include a new API for doing new things, you increase the attack surface. DirectX is, as with HTML and XAML, largely included complete within Windows Store apps.

The Windows Store Sandbox

The sandbox for Windows Store apps is really no different than the sandbox that iOS, Android, Java or any other runtime application runs in. The only tweak is that there are three platforms (as shown in Figure 1), driving a much larger attack surface.

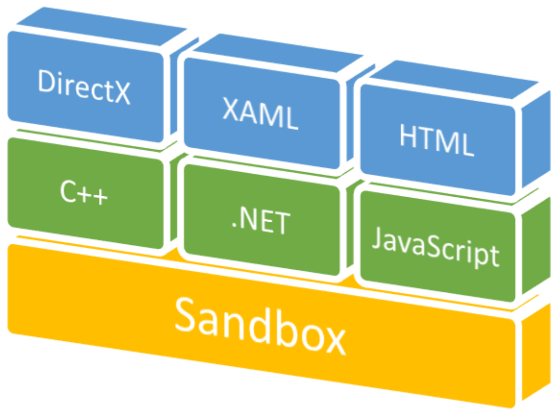


Figure 1: The Windows Store Sandbox

Although there is a large set of vectors for attack here, there was a significant and apparently successful effort to harden the sandbox. It is very difficult, if not impossible, to leak from the sandbox into the operating system. As such, most of the testing and hardening of application logic is made up of information protection and attacking the underlying services.

Testing Windows Store apps for application security

Because of the relative strength of the sandbox, most Windows Store app security testing will focus on the backend services, and flaws in the business logic of the application.

This isn't to say that the sandbox is impenetrable. I am sure there are those who will find flaws. I am a blue team person, and I abide by the 80/20 rule. I want to fix the 20% of the code that will keep out 80% of the attackers.

Backend Services

I profile and test the services that support an application using Zed Attack Proxy.

Setting up ZAP for Windows 8

After installing ZAP, it needs to be made ready to act as a proxy for Metro applications. As most Metro applications make heavy use of the Internet for data storage, information access and communication with the Windows Store, use of a proxy is a great place to start to test the underlying security.

1) Press the Windows button and type Internet Options.

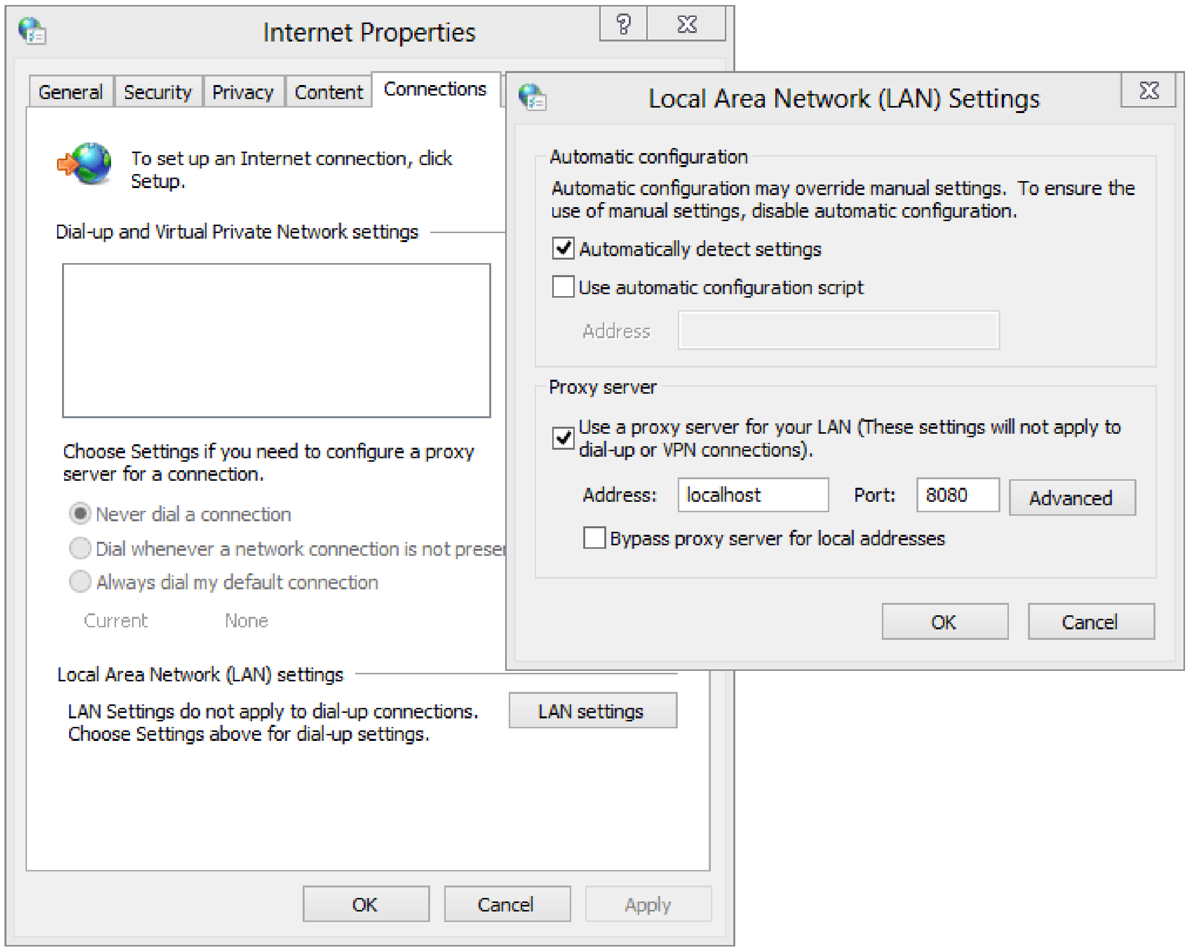
2) Press the Settings tab on the right of the Start Screen.

3) Select the Internet Options control panel.

4) Click the Connections tab.

5) Click the LAN Settings button

6) Change the Proxy Server settings like Figure 1. Check the 'Use a proxy server' checkbox, then set the address to 'localhost' and the port to '8080'.



To set up an Internet connection, click Setup.

Dial-up and Virtual Private Network settings

Choose Settings if you need to configure a proxy server for a connection.

- [✓] Never dial a connection
- Dial whenever a network connection is not presei
- Always dial my default connection

Current None

Local Area Network (LAN) settings

LAN Settings do not apply to dial-up connections. Choose Settings above for dial-up settings.

Automatic configuration

Automatic configuration may override manual settings. To ensure the use of manual settings, disable automatic configuration.

- [✓] Automatically detect settings
- [ ] Use automatic configuration script

Address

Proxy server

- [✓] Use a proxy server for your LAN (These settings will not apply to dial-up or VPN connections).

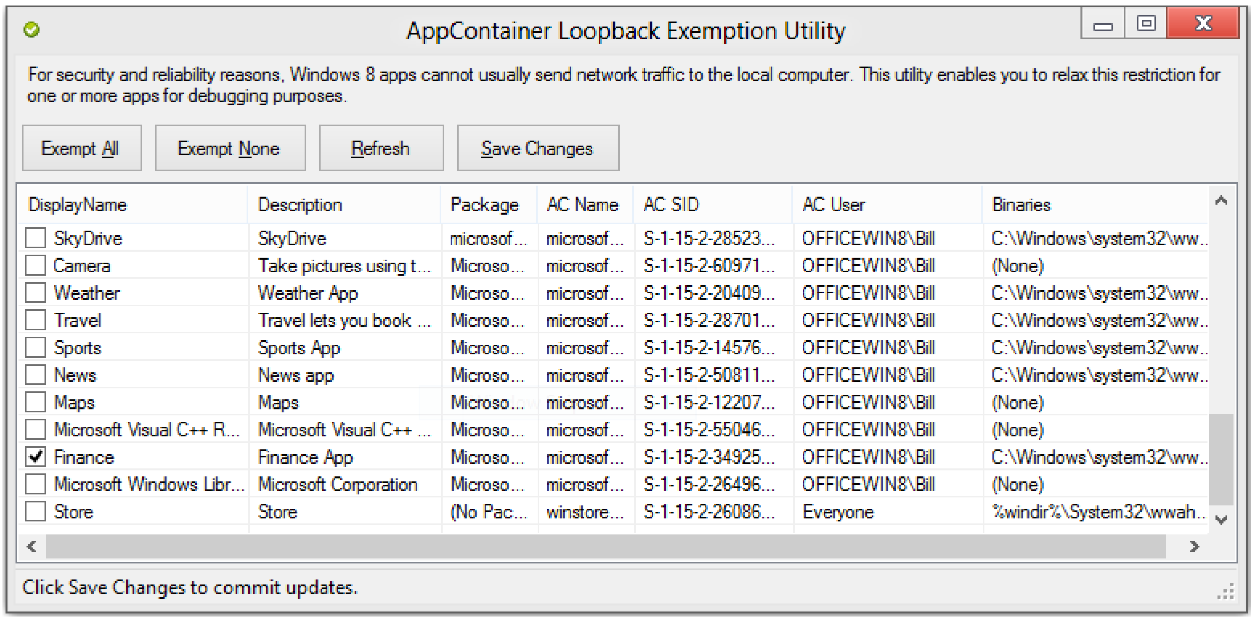
Address: localhost Port: 8080

- [ ] Bypass proxy server for local addresses

Figure 1 - configuring a proxy

7) Click OK, then click OK.

Finally, you have to configure the app you are testing to allow the loopback (localhost) address as a proxy, which by default AppContainers aren't allowed to do. You can do this with PowerShell, but it is a lot easier to download Eric Law's excellent EnableLoopback utility, which will do the work for you. You just need to find the app you want to test and select the checkbox, then click Save Changes, like Figure 2.



AppContainer Loopback Exemption Utility

For security and reliability reasons, Windows 8 apps cannot usually send network traffic to the local computer. This utility enables you to relax this restriction for one or more apps for debugging purposes.

DisplayName	Description	Package	AC Name	AC SID	AC User	Binaries
☐ SkyDrive	SkyDrive	microsof...	microsof...	S-1-15-2-28523...	OFFICEWIN8\Bill	C:\Windows\system32\ww...
☐ Camera	Take pictures using t...	Microso...	microsof...	S-1-15-2-60971...	OFFICEWIN8\Bill	(None)
☐ Weather	Weather App	Microso...	microsof...	S-1-15-2-20409...	OFFICEWIN8\Bill	C:\Windows\system32\ww...
☐ Travel	Travel lets you book ...	Microso...	microsof...	S-1-15-2-28701...	OFFICEWIN8\Bill	C:\Windows\system32\ww...
☐ Sports	Sports App	Microso...	microsof...	S-1-15-2-14576...	OFFICEWIN8\Bill	C:\Windows\system32\ww...
☐ News	News app	Microso...	microsof...	S-1-15-2-50811...	OFFICEWIN8\Bill	C:\Windows\system32\ww...
☐ Maps	Maps	Microso...	microsof...	S-1-15-2-12207...	OFFICEWIN8\Bill	(None)
☐ Microsoft Visual C++ R...	Microsoft Visual C++ ...	Microso...	microsof...	S-1-15-2-55046...	OFFICEWIN8\Bill	(None)
✓ Finance	Finance App	Microso...	microsof...	S-1-15-2-34925...	OFFICEWIN8\Bill	C:\Windows\system32\ww...
☐ Microsoft Windows Libr...	Microsoft Corporation	Microso...	microsof...	S-1-15-2-26496...	OFFICEWIN8\Bill	(None)
☐ Store	Store	(No Pac...	winstore...	S-1-15-2-26086...	Everyone	%windir%\System32\wwah...

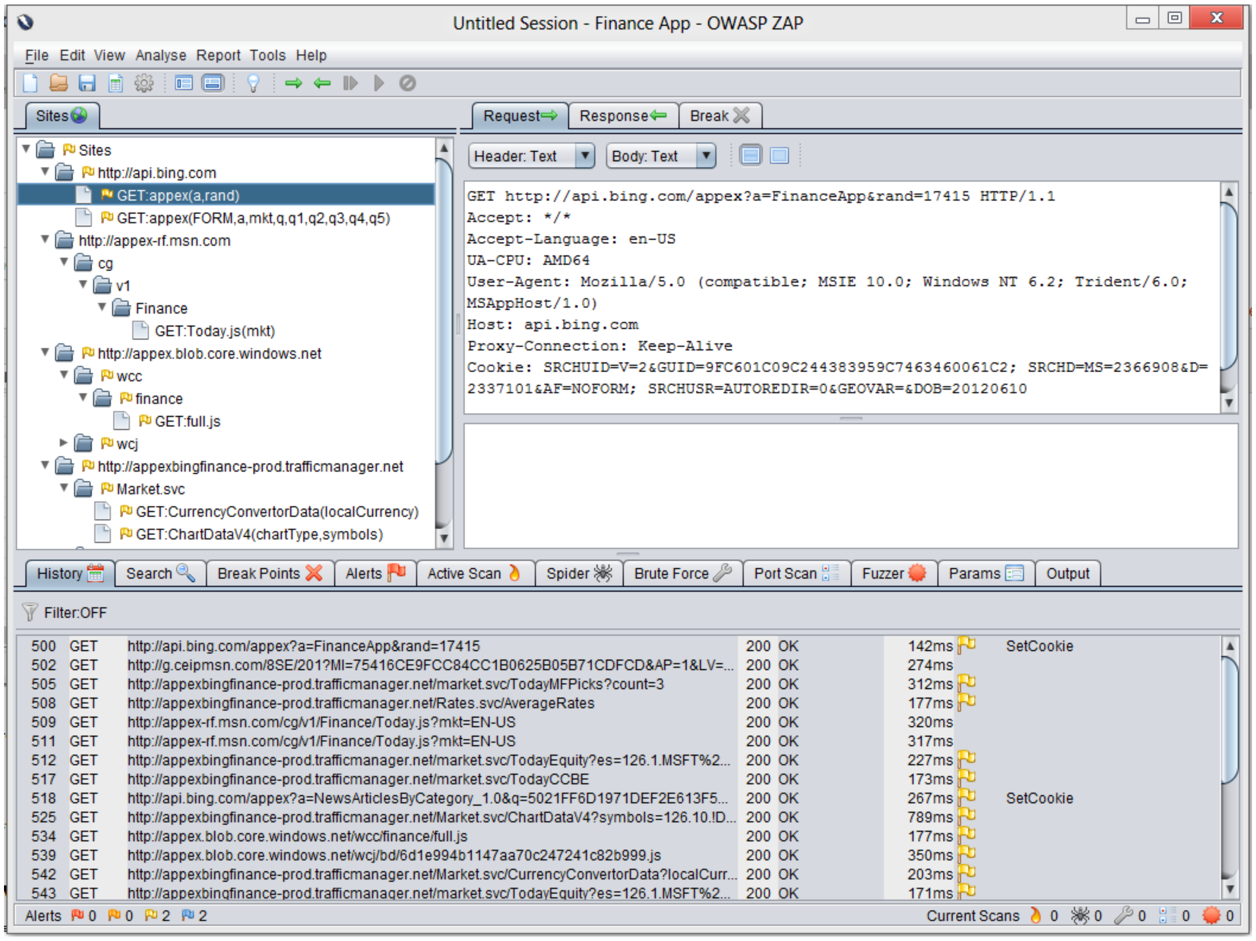
Click Save Changes to commit updates.

Figure 2: the EnableLoopback Utility

Not your machine is configured to use ZAP as a proxy. While these settings are set, you will have to have ZAP running to use the Internet. Also, enabling the loopback address circumvents important security controls placed on Metro apps. Only use it on a test system.

Testing your settings

To test your settings, run ZAP by pressing the Windows key and typing ZAP then pressing enter. Then run an installed Metro app, like the Finance app, which you have configured for loopback. In ZAP, all of the services being called by the application will appear in the Sites pane, and all of the individual calls will appear in the History tab. Figure 3 shows my results.



```
Sites
  http://api.bing.com
    GET:appex(a,rand)
    GET:appex(FORM,a,mkt,q,q1,q2,q3,q4,q5)
  http://appex-rf.msn.com
    cg
      v1
        Finance
          GET:Today.js(mkt)
  http://appex.blob.core.windows.net
    wcc
      finance
        GET:full.js
    wcj
  http://appexbingfinance-prod.trafficmanager.net
    Market.svc
      GET:CurrencyConvertorData(localCurrency)
      GET:ChartDataV4(chartType,symbols)
```

```
GET http://api.bing.com/appex?a=FinanceApp&rand=17415 HTTP/1.1
Accept: */*
Accept-Language: en-US
UA-CPU: AMD64
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0;
MSAppHost/1.0)
Host: api.bing.com
Proxy-Connection: Keep-Alive
Cookie: SRCHUID=V=2&GUID=9FC601C09C244383959C7463460061C2; SRCHD=MS=2366908&D=
2337101&AF=NOFORM; SRCHUSR=AUTOREDIR=0&GEOVAR=&DOB=20120610
```

```
500  GET  http://api.bing.com/appex?a=FinanceApp&rand=17415                                        200 OK  142ms  SetCookie
502  GET  http://g.ceipmsn.com/8SE/201?MI=75416CE9FCC84CC1B0625B05B71CDFCD&AP=1&LV=...              200 OK  274ms
505  GET  http://appexbingfinance-prod.trafficmanager.net/market.svc/TodayMFPicks?count=3          200 OK  312ms
508  GET  http://appexbingfinance-prod.trafficmanager.net/Rates.svc/AverageRates                   200 OK  177ms
509  GET  http://appex-rf.msn.com/cg/v1/Finance/Today.js?mkt=EN-US                                200 OK  320ms
511  GET  http://appex-rf.msn.com/cg/v1/Finance/Today.js?mkt=EN-US                                200 OK  317ms
512  GET  http://appexbingfinance-prod.trafficmanager.net/market.svc/TodayEquity?es=126.1.MSFT%2...  200 OK  227ms
517  GET  http://appexbingfinance-prod.trafficmanager.net/market.svc/TodayCCBE                     200 OK  173ms
518  GET  http://api.bing.com/appex?a=NewsArticlesByCategory_1.0&q=5021FF6D1971DEF2E613F5...        200 OK  267ms  SetCookie
525  GET  http://appexbingfinance-prod.trafficmanager.net/Market.svc/ChartDataV4?symbols=126.10.!D... 200 OK  789ms
534  GET  http://appex.blob.core.windows.net/wcc/finance/full.js                                   200 OK  177ms
539  GET  http://appex.blob.core.windows.net/wcj/bd/6d1e994b1147aa70c247241c82b999.js              200 OK  350ms
542  GET  http://appexbingfinance-prod.trafficmanager.net/Market.svc/CurrencyConvertorData?localCurr... 200 OK  203ms
543  GET  http://appexbingfinance-prod.trafficmanager.net/market.svc/TodayEquity?es=126.1.MSFT%2...  200 OK  171ms
```

```
Alerts 0 0 2 2                                         Current Scans 0 0 0 0 0
```

Figure 3 - Finance Test results

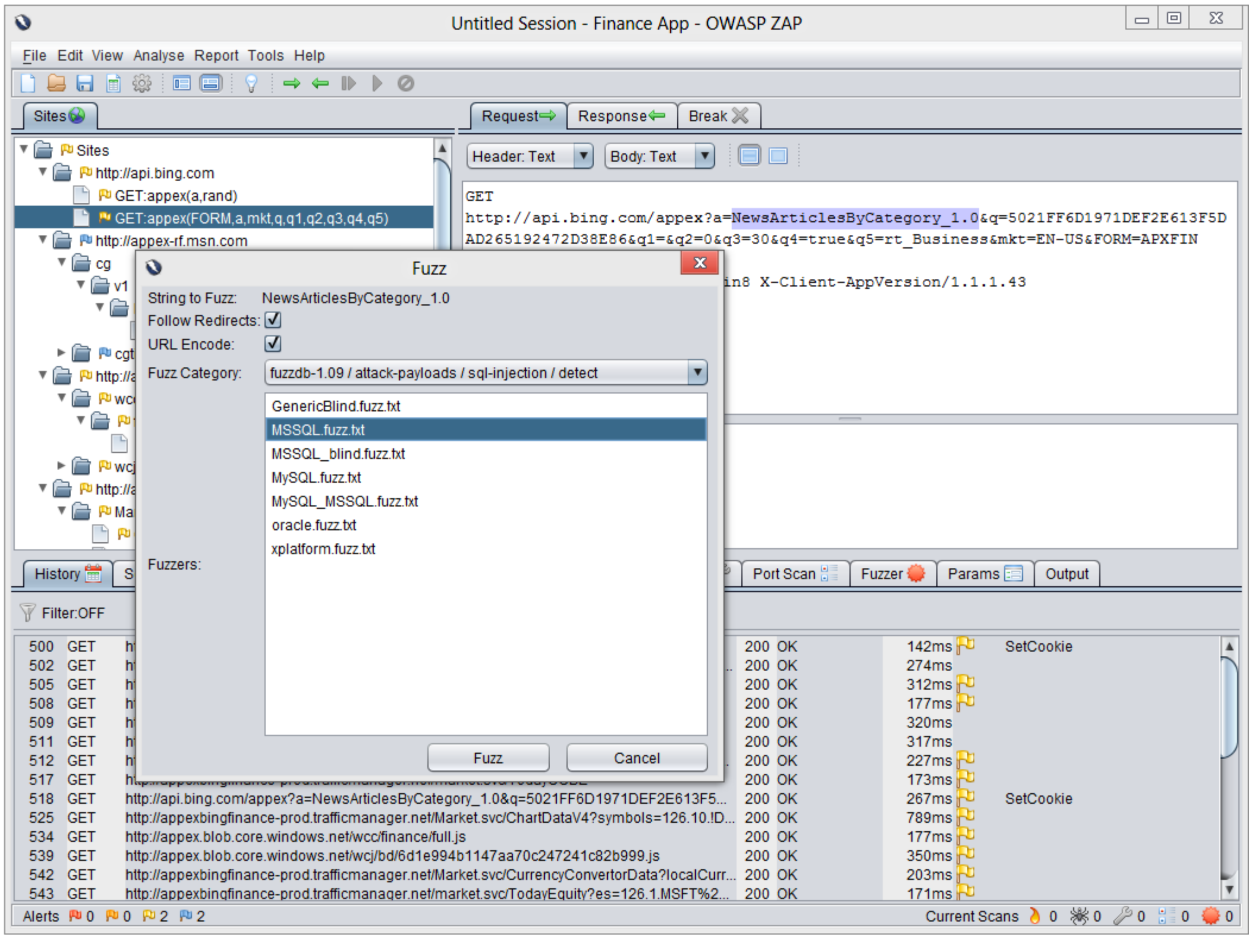
Each of these represents an HTTP call being made by the Finance app to populate its screens. Apparently, the developers of this app chose not to use SSL for any of the calls, which makes our job easier - although since we have a man in the middle the analysis can still be carried forward. Either way, this gives us a ripe field to begin our analysis.

Attacking underlying services

You can now use ZAP to test the underlying services with the fuzzer.

1) Select a service to test in the Sites pane. I selected the api.bing.com/appex service.

2) Highlight a parameter to fuzz in the Request pane, right in the querystring. I selected the 'a' parameter in Figure 4.



```
GET
http://api.bing.com/appex?a=NewsArticlesByCategory_1.0&q=5021FF6D1971DEF2E613F5D
AD265192472D38E86&q1=&q2=0&q3=30&q4=true&q5=rt_Business&mkt=EN-US&FORM=APXFIN

in8 X-Client-AppVersion/1.1.1.43
```

Sites
- http://api.bing.com
  - GET:appex(a,rand)
  - GET:appex(FORM,a,mkt,q,q1,q2,q3,q4,q5)
- http://appex-rf.msn.com

String to Fuzz: NewsArticlesByCategory_1.0
Follow Redirects: ✓
URL Encode: ✓
Fuzz Category: fuzzdb-1.09 / attack-payloads / sql-injection / detect
Fuzzers:
- GenericBlind.fuzz.txt
- MSSQL.fuzz.txt
- MSSQL_blind.fuzz.txt
- MySQL.fuzz.txt
- MySQL_MSSQL.fuzz.txt
- oracle.fuzz.txt
- xplatform.fuzz.txt

```
500  GET  h…   200 OK   142ms   SetCookie
502  GET  h…   200 OK   274ms
505  GET  h…   200 OK   312ms
508  GET  h…   200 OK   177ms
509  GET  h…   200 OK   320ms
511  GET  h…   200 OK   317ms
512  GET  h…   200 OK   227ms
517  GET  h…   200 OK   173ms
518  GET  http://api.bing.com/appex?a=NewsArticlesByCategory_1.0&q=5021FF6D1971DEF2E613F5...  200 OK  267ms  SetCookie
525  GET  http://appexbingfinance-prod.trafficmanager.net/Market.svc/ChartDataV4?symbols=126.10.!D...  200 OK  789ms
534  GET  http://appex.blob.core.windows.net/wcc/finance/full.js  200 OK  177ms
539  GET  http://appex.blob.core.windows.net/wcj/bd/6d1e994b1147aa70c247241c82b999.js  200 OK  350ms
542  GET  http://appexbingfinance-prod.trafficmanager.net/Market.svc/CurrencyConvertorData?localCurr...  200 OK  203ms
543  GET  http://appexbingfinance-prod.trafficmanager.net/market.svc/TodayEquity?es=126.1.MSFT%2...  200 OK  171ms
```

Figure 4 - Setting the fuzzing dictionary

3) Right click and select Fuzz. The dialog that appears is also shown in Figure 4.

4) Select the Fuzz category in the dropdown. I selected SQL Injection in Figure 4.The list of pre-installed fuzzers appear in the Fuzzers listbox. These are known attacks that work against a variety of databases. As this is a Microsoft service, we can assume that MSSQL is probably the database, although other test tools could be used to ascertain this. In Figure 4 that is the fuzzer I selected.

5) Click Fuzz to begin your attack. ZAP will begin to call the service with the attacks in the SQL Injection file.

And ... it didn't work at all. The Fuzzer tab shows each attack, and all of them have a 404 error in the result column. Apparently that field is being used as a controller method name - I should have thought of that!

Oh well, let's try something else. Select the 'q' parameter and try again with a SQL attack. This one is a lot better - every result comes back as a 200. Awesome. Click on a line in the Fuzzer tab to see the result in the Response tab.

```
{
   "BdiGeneric_BingResponse_1_0":{
      "AppNS":"AppEx",
      "Responses":[
      ],
      "Errors":[
         {
            "Code":1002,
            "Message":"Parameter has invalid value.",
            "Parameter":"AppExNewsVerticalRequest.AppId",
            "Value":"'create user name identified by pass123 temporary tablespace temp default tablespace users; "
         }
      ]
   }
}
```

What have we here? An error code 1002? Aah, 'Parameter has invalid value.' -- they are using parameterized queries. I suppose one would expect a well vetted service like Bing to be hardened against the #1 exploit on the OWASP Top 10 though.

Code Review

The second most significant flaw is bad programming. I am sure you are just shocked by this turn of events but it is true. It is just as possible to build an insecure application with bad code in Windows 8 as it is in a web application on any platform. Especially if you are working in HTML5.

Let's start there, with simple Cross Site Scripting flaws, and then move on to an optimistic eval() statement.

XSS

Cross site scripting is a common flaw in web applications, but it is just as possible in a Windows Store application. As usual, the key is sanitizing your inputs. Code Review should catch flaws like this, but it requires a careful and thoughtful review.

Bear in mind that the IE10 rendering engine does not allow basic XSS. Take the zen example – we have a site that accepts a value, stores it in roamingSettings and then presents it on the screen on next load. The HTML just looks like this:

```
<body>
       <div  style="padding: 30px;">
              <H1>Bill's XSSer</H1>
              <p> This is something new</p>
              <p>Enter the value:<input type="text" id="theValue">
                     <button id="theButton">Submit</button>
              </p>
              <p>Here is the previous value: <span id="previousValue"></span></p>
       </div>
</body>
```

The JavaScript that runs the site looks like this:

```
(function () {
    "use strict";

    WinJS.Binding.optimizeBindingReferences = true;

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;
    var roamingSettings = Windows.Storage.ApplicationData.current.roamingSettings;


    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            args.setPromise(WinJS.UI.processAll().then(function () {
                var remoteValue = roamingSettings.values['theValue1'];
                document.getElementById('previousValue').innerHTML = remoteValue;
                document.getElementById('theButton').addEventListener('click', function () {
                    roamingSettings.values['theValue1'] = document.getElementById("theValue").value;
                });
            }));
        }
    };
    app.start();
})();
```

Now if we run the app and enter the string '';!--"<XSS>=&{()} the result is:

```
HTML1701: Unable to add dynamic content '';!--"<XSS>=&{()}'. A script
attempted to inject dynamic content, or elements previously modified
dynamically, that might be unsafe. For example, using the innerHTML property
to add script or malformed HTML will generate this exception. Use the
toStaticHTML method to filter dynamic content, or explicitly create elements
and attributes with a method such as createElement. For more information, see
http://go.microsoft.com/fwlink/?LinkID=247104.

File: default.html
```

Where does that leave us? Is it not injectable? Of course not. There is always a solution. But this is a defense talk and simply to prevent this kind of attack, you should never leave scriptable elements on the page. The flaw is use of the innerHTML property of the span 'previousValue'. Instead, innerText should be used.

From the code review perspective, anytime user input is handled on either end, it needs to be reviewed. Sanitize inputs and outputs. It's not that tough. The kicker is that it is currently VERY tough to smoke test a Windows 8 application. The testing I did for this sample was ALL manual, copying and pasting from RSnake's XSS cheatsheet.

Optimistic eval and other bad code

Performing an Eval with data that is provided by a user in the payload is always a bad idea, but it can create utter havoc in a cloud based Windows 8 application. Fortunately, eval and execUnsafeLocalFunction (both supported in Windows 8) are easy to search for in the codebase.

WinRT features

There are a number of WinRT features that are potential attack vectors. You need to consider their application, and determine their risk individually based on the use factor.

Identity

Generally, your login for Windows 8 system not on a domain is your Microsoft Account, formerly Live or Passport login. For many users this is the key to a lot of different services, like HotMail, SkyDrive, MSDN and TechNet for technicians, social networking and more. The WinRT API gives a developer an easy way to leverage this information with Live Connect.

The Live Connect Identity API takes advantage of this always logged in state to allow apps to integrate with the user's account information, given permission. In HTML, you simply provide a named div tag, and initialize it in the JavaScript.

```
WL.ui({
    name: "signin",
    element: "signin"
});
```

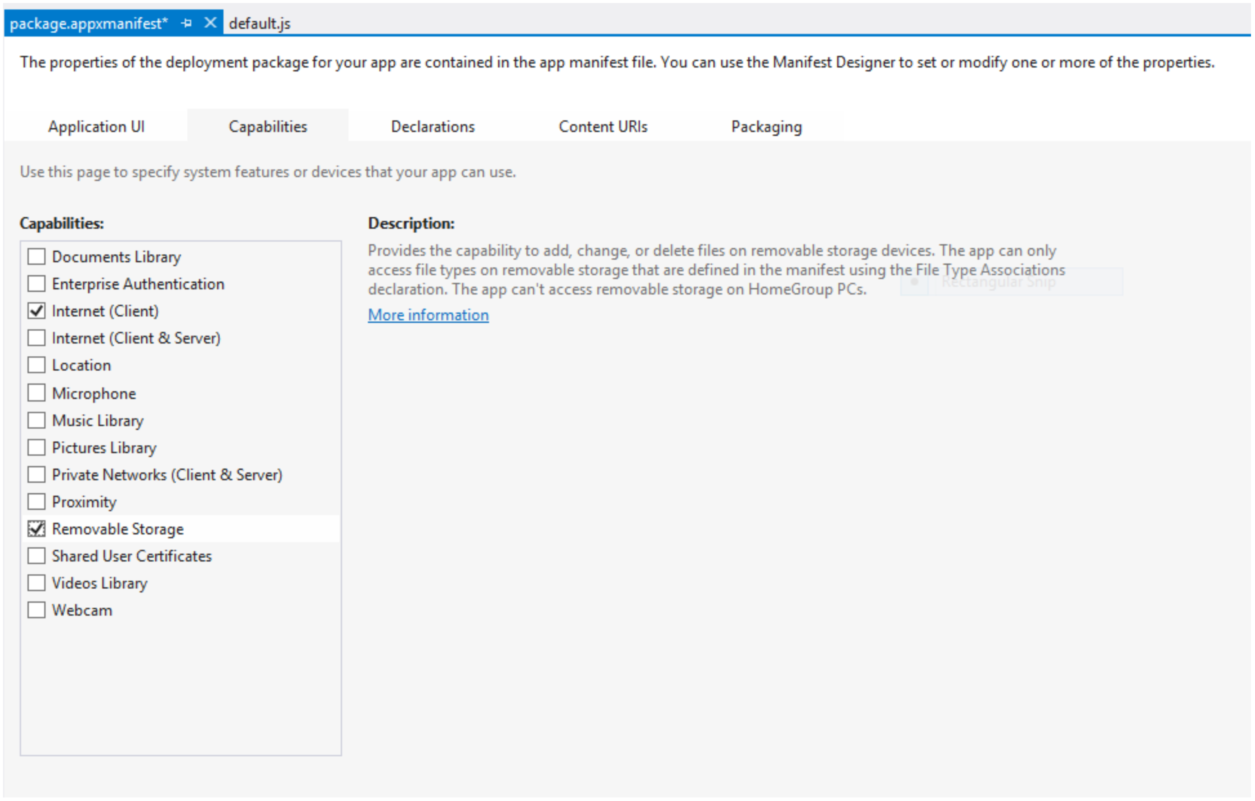
Once the control is in place, the WL namespace provides a simple login function that adds the Windows login experience to your app. The login method is an async call that passes the credentials to a callback function, from which you can get user information like name and profile photo.

```
function signInUser() {
    WL.login({
        scope: "wl.signin"
    }, onSignIn);
}
```

While the Live Connect service is an awesome feature, it seems like a very ripe CSRF target to me. Given that this isn't something you need to harden your apps about, but the ease with which a malicious app might be able to make use of users' credentials seems like something to take into consideration when setting up the policy for your organization.

Capabilities

In order to use features of the Windows 8 operating system, you need to declare capabilities your app will use. This is not unsimilar to what the iOS and Android operating system do. When you declare a requirement in the capabilities as shown in Figure 5, the user will be notified that the app will be using that feature of their device on install and at runtime.



The properties of the deployment package for your app are contained in the app manifest file. You can use the Manifest Designer to set or modify one or more of the properties.

Use this page to specify system features or devices that your app can use.

Capabilities:

- ☐ Documents Library
- ☐ Enterprise Authentication
- ☑ Internet (Client)
- ☐ Internet (Client & Server)
- ☐ Location
- ☐ Microphone
- ☐ Music Library
- ☐ Pictures Library
- ☐ Private Networks (Client & Server)
- ☐ Proximity
- ☑ Removable Storage
- ☐ Shared User Certificates
- ☐ Videos Library
- ☐ Webcam

Description:

Provides the capability to add, change, or delete files on removable storage devices. The app can only access file types on removable storage that are defined in the manifest using the File Type Associations declaration. The app can't access removable storage on HomeGroup PCs.

More information

Figure 5: Let's use the USB Drive. That's a good idea, right?

Capabilities are divided into General and Special use. General use expands the functionality of an app to include libraries and devices. Special use capabilities are the Windows Store equivalent of administrative rights. Least privilege is the rule.

The least understood of these are the Documents Library and Enterprise Authentication. The Document library is not necessary to access files in the documents library. Counterintuitive, I know. Documents Library is used for access to file extensions, and you probably don't need it. Don't register it if you don't need it.

The Enterprise Authentication capability is not used to log into things, that's what the credential picker is for. Enterprise Authentication is used to access enterprise resources programmatically. You probably don't need it so don't use it.

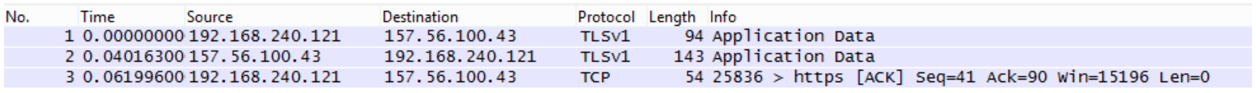
Remote Storage

This one freaks me out a bit, even though I haven't found a way to exploit it. A developer can use Windows.Storage.ApplicationData.current.roamingSettings to save values 'somewhere' that a user can then pick up in the same app in the same user account on another machine.

If you look at the example I used for the XSS testing above, I used roaming settings then. When I went to test on my laptop, I checked out the code from subversion, ran it, and got the error because the XSS test was still in the roaming settings. I needed to change the variable name to get the app to run.

Anyway, this is clearly sending stuff to Azure and storing it with my Microsoft account and app ID. I am sure it is all secure and stuff. Don't you just think it's strange that this data is just going somewhere, where we have no control at all, and then getting pulled back for later use?

Looking at WireShark, you can see where the send and receive happen to 157.56.100.43.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.00000000	192.168.240.121	157.56.100.43	TLSv1	94	Application Data
2	0.04016300	157.56.100.43	192.168.240.121	TLSv1	143	Application Data
3	0.06199600	192.168.240.121	157.56.100.43	TCP	54	25836 > https [ACK] Seq=41 Ack=90 Win=15196 Len=0

Figure 6: Wireshark results for remoteSettings

A man in the middle attack can get access to the TLS data, but either way, isn't there a privacy consideration storing my apps data who knows where?

Local Storage

The local storage in Windows.Storage isn't that much of a big deal – apps keep local data all the time. It's the other kinds of local storage that I am concerned with – that which is kept in HTML5 storage.

SessionStorage, LocalStorage and WebSQL all work in Windows Store apps. They all make use of a database that is stored locally and can be accessed easily with the Windows Explorer.

In general, watch how you store your data. Services can be peeked at, remote settings are going who-knows-where and local HTML5 databases are all editable by the user.

Networking

It is interesting the constraints that are put on the Windows Store apps related to networking. In .NET, the System.Net namespace contains very comprehensive networking components. In WinRT's .NET, however, there is nothing but web access.

There is a reason for this. The Windows RT sandbox has some significant constraints when it comes to networking. For instance, since there is no outgoing socket connection, you can't connect to a database. There is no ADO.NET in Windows Store apps. There is just HTTP.

That means that you have to use services to get everything into and out of your database. That takes us back to testing services very well, because the attack surface is broadening and broadening.

Countermeasures provided by WinRT

There are quite a few features within Windows RT that will allow you to lock down portions of your app. Authentication, encryption and exception handling are just a few of the bits that are still maturing, but a significant part of the development environment.

Encryption and Hashing with Windows.Security

Windows.Security has some hashing and encryption APIs, replacing the .NET libraries that do the same thing. They are slow, but they are effective.

If you are going to communicate with a backend service, hashing the authentication information is the best way to prevent a man in the middle attack from gaining access to the credentials. WinRT has algorithms up through SHA512 via Windows.Security.Cryptography.Core.

In order to hash, you need to encode and decode using the CryptographicBuffer class, and it is not fast. Here is a look at the code that you might use to hash to SHA512:

```
function hashWithSha512(value) {
    var crypto = Windows.Security.Cryptography;
    var encodedString = crypto.CryptographicBuffer.convertStringToBinary(value, crypto.BinaryStringEncoding.utf8);
    var algorithmProvider = new crypto.Core.HashAlgorithmProvider.openAlgorithm(crypto.Core.HashAlgorithmNames.sha512);
    var hashedValue = algorithmProvider.hashData(encodedString);
    return crypto.CryptographicBuffer.encodeToHexString(hashedValue);
}
```

This really applies when trying to authorize to a service. If you send the credentials over the wire, a simple analysis with ZED (as shown above) will garner the attacker your credentials. Using a hashed value made up of a series of different data points (say the device identifier, the current time and some salt) can manage session level identification for a service call. You can get the deviceID from Windows.System.Profile.HardwareIdentification.getPackageSpecificToken.

Another use of this is credential evaluation, where the service accepts a hash of the password along with some session specific salt for authentication.

Identity with OAuth

The Windows API includes OAUTH built in. This is better than having everyone bake their own, but still is something of an attack surface. Also, you should know that there is no need to create a user repository in your systems – let Facebook or Twitter handle it for you.

Here is what it looks like:

```
var startURI = new Windows.Foundation.Uri(facebookURL);
var endURI = new Windows.Foundation.Uri(callbackURL);

Windows.Security.Authentication.Web.WebAuthenticationBroker.authenticateAsync(
       Windows.Security.Authentication.Web.WebAuthenticationOptions.default,
       startURI,
       endURI).done(function (result) {
                    //do something});
```

Kernel? No thanks.

Ken Johnson and Matt Miller did an awesome job of talking kernel hardening in Abu Dhabii last year, so I will defer to them for specifics. Let it be said that most of the kernel faults that contemporary malware takes advantage of as been mitigated in Windows 8. Sinofsky gives a great overview of the details in a blog post outlining Johnson and Miller's talk:

> *Address Space Layout Randomization (ASLR). ASLR was first introduced in Windows Vista and works by randomly shuffling the location of most code and data in memory to block assumptions that the code and data are at same address on all PCs. In Windows 8, we extended ASLR's protection to more parts of Windows and introduced enhancements such as increased randomization that will break many known techniques for circumventing ASLR.*
>
> *Windows kernel. In Windows 8, we bring many of the mitigations to the Windows kernel that previously only applied to user-mode applications. These will help improve protection against some of the most common type of threats. For example, we now prevent user-mode processes from allocating the low 64K of process memory, which prevents a whole class of kernel-mode NULL dereference vulnerabilities from being exploited. We also added integrity checks to the kernel pool memory allocator to mitigate kernel pool corruption attacks.*
>
> *Windows heap. Applications get dynamically allocated memory from the Windows user-mode heap. Major redesign of the Windows 8 heap adds significant protection in the form of new integrity checks to help defend against many exploit techniques. In addition, the Windows heap now randomizes the order of allocations so that exploits cannot depend on the predictable placement of objects—the same principle that makes ASLR successful. We also added guard pages to certain types of heap allocations, which helps prevent exploits that rely on overrunning the heap.*
>
> *Internet Explorer. "Use-after-free" vulnerabilities represented nearly 75% of the vulnerabilities reported in Internet Explorer over the last two years. For Windows 8, we implemented guards in Internet Explorer to prevent an attacker from crafting an invalid virtual function table, making these attacks more difficult. Internet Explorer will also take full advantage of the ASLR improvements provided by Windows 8.*

I have yet to see a decent low-level vulnerability exploited in Windows 8 Store apps. We are far from suggesting that the risk is gone, but it is certainly much less than in previous versions.

Unexpected behavior just crashes the app

When the app runs into something unexpected (like an XSS attack) it just crashes. There is an error handler of last resort in WinJS that just gives you a last chance with a debugger, and then kills the app outright.

```
var terminateAppHandler = function (data, e) {
    // This is the unhandled exception handler in WinJS. This handler is invoked whenever a promise
    // has an exception occur that is not handled (via an error handler passed to then() or a call to done()).
    //
    // To see the original exception stack, look at data.stack.
    // For more information on debugging and exception handling go to http://go.microsoft.com/fwlink/p/?LinkId=253583.
    debugger;
    MSApp.terminateApp(data);
};
```

As a developer, this is supremely frustrating. As a consumer, this is a good thing. If a malicious app shows up, it is much more likely to just bomb when it does something unexpected than it is to do damage, especially if I reject a capability that the application is expecting (as I should). Of course, there isn’t much that Windows can do when the user accepts everything, but we are taking this one step at a time.

General good ideas in Windows Store app development

So what do we do? Here are a few general guidelines that I follow.

Coding defensively

They can see your code. Anyone who installs an app can see the HTML and JS, or in .NET at least the XAML (although dotPeek will give you a pretty good look at the binary as well).

Getting to the code

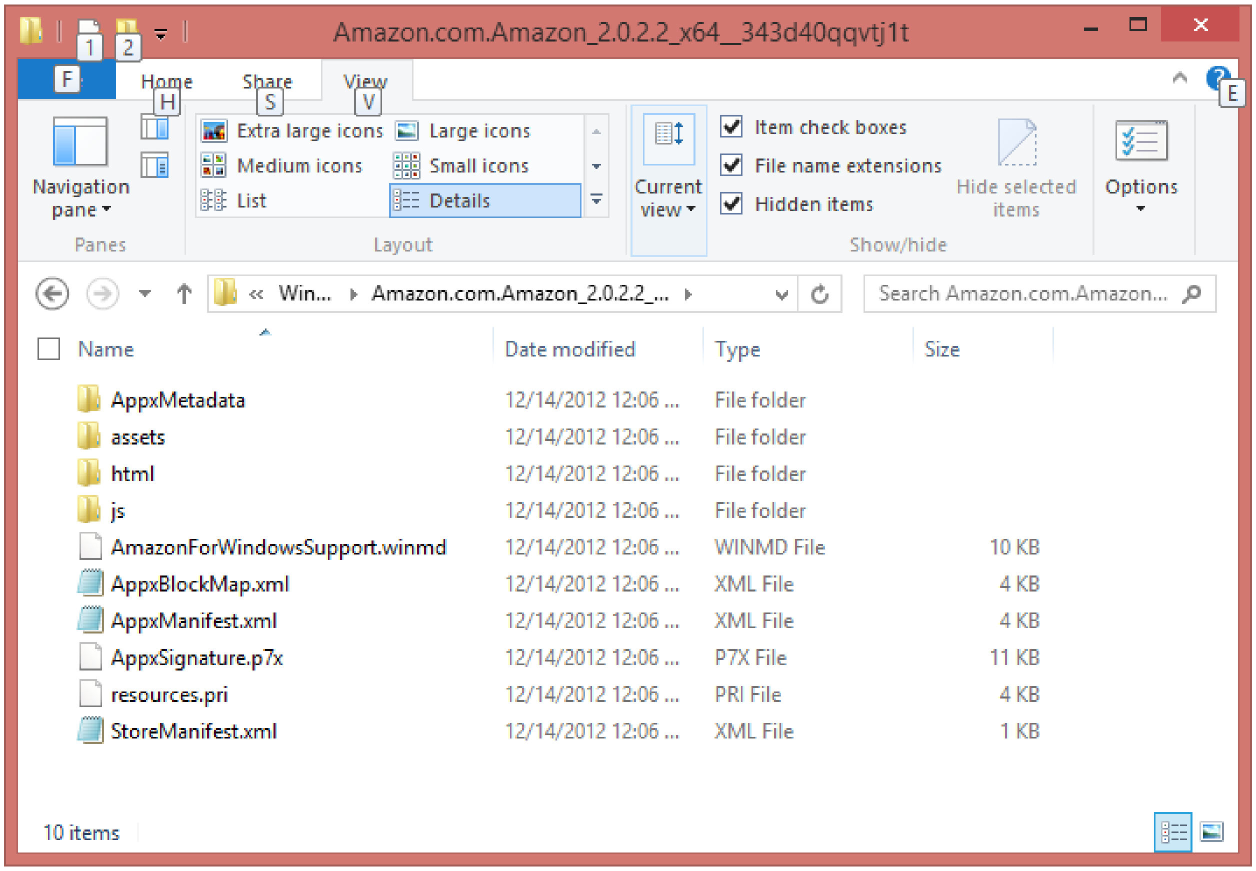
Files for installed apps are in C:\Program Files\WindowsApps, which is 'hidden' so make sure you have 'View Hidden Items' turned on. This folder isn't accessible by the governing user even if you are running as the administrator, which you shouldn't be. The easiest way to get access to the folder is to take ownership of the folder and everything in it. You can do this with Powershell if you like.

```
Get-ChildItem 'C:\Program Files\WindowsApps' -recurse | ForEach-Object {Get-Acl $_.FullName}
```

But is remarkably easy to do in the UI, so that is how I did it. Right click on C:\Program Files\WindowsApps and select Properties, then click on the Security tab. From there you can click Advanced to get the Advanced Security Settings.

Click the Change link and then type in your Microsoft Account email address. Then press enter, OK, and OK again to close all of the dialog boxes. From there you can easily navigate to any installed apps.

Open one of these, Amazon for instance, and you can see that the file structure is just as it would be in Visual Studio.



Name	Date modified	Type	Size
AppxMetadata	12/14/2012 12:06 ...	File folder	
assets	12/14/2012 12:06 ...	File folder	
html	12/14/2012 12:06 ...	File folder	
js	12/14/2012 12:06 ...	File folder	
AmazonForWindowsSupport.winmd	12/14/2012 12:06 ...	WINMD File	10 KB
AppxBlockMap.xml	12/14/2012 12:06 ...	XML File	4 KB
AppxManifest.xml	12/14/2012 12:06 ...	XML File	4 KB
AppxSignature.p7x	12/14/2012 12:06 ...	P7X File	11 KB
resources.pri	12/14/2012 12:06 ...	PRI File	4 KB
StoreManifest.xml	12/14/2012 12:06 ...	XML File	1 KB

Figure 7: Source of Amazon's Windows Store app

From there you can browse at will. As expected, apps written in C++ or .NET are compiled EXEs, although the XAML is still browsable. Nothing that a small dose of dotPeek won't fix, I am sure. I'll leave that as an exercise for the reader.

The implications

What are the implications of all of this? Not a lot really. For most apps written in JavaScript, there is nothing there that a quick right-click, browse-source won't do for you on that organization's web site. JavaScript is just as discoverable as HTML in those cases.

For some apps, though, there may be some consideration for keys to services such as Google, Bing or other service providers. For instance, we are using Weather Underground in an app I am developing, and the client's use key is right there in the settings.js file. Usually, that would be accessed server-side in a regular web app, and apparently we are going to have to do something similar for the Windows 8 app. Perhaps wrapping external services in your own service layer and then accessing those to get the values provided by the original service.

There has been some discussion of changing code, recompiling using the provided key, and getting access to for pay content or disabling ads. I am not a fan of ripping of developers (being one myself) so I won't go into that here. It could be a very significant detriment to those seeking to monetize apps, however.

The most significant thing, as usual, is developer ignorance. If the developer doesn't realize that all of the code is easily browsable, then they may make poor decisions in the construction, commenting or production of the code base.

Choosing your storage very carefully

Comparing web sites to Windows Store apps is a common pastime. There is a lot in common between the two, and in fact there is a large debate between building mobile apps and mobile HTML5 sites. One significant consideration is the dependence on a backend server to store significant data on. Web sites have the benefit of being rendered on a server. Windows Store apps do not.

Take for instance the consumption of paid content: weather, let's say. In a web application, the server takes care of communicating with the weather service and rendering the details to something viewable for the user. The user never gets to see the details. They just get to see the weather.

In a Windows Store app, however, there is no server. The app has to call and get the weather from the client machine. If there is a 'secret', which there nearly always is, it is compromised two ways mentioned above. The source code is visible to the user, and the traffic to the service can be easily viewed.

This isn't the only client side storage problem, either. As mentioned in remote storage and local storage above, keeping sensitive data on the machine and sending it up to Azure both seem like questionable ideas. The local databases are fairly easy to access, and the remote storage is a black box.

Your app will have to store something sometime, but there is no question that keeping as little as possible in storage is a good idea. If you need a lot of data to go in and out, make your own service layer using Heroku or Azure Mobile Services, and keep it encrypted. At the very least, review your storage needs with the knowledge that much is quite open to being compromised.

Testing your service layer

In a similar vein, no matter how you implement your service layer, consider carefully how you use it. Develop carefully, use it with an eye toward the realization that nothing is private, and test furiously with a fuzzing tool.

If you are building your own service backend, consider carefully how you will handle authentication and authorization. It isn't a good idea to handle authentication using just a token, as it can be intercepted and reused. Remember that service communication with a Window Store app is just like form communication with a web site – they user isn't necessarily constrained by the user interface.

On the other hand, consider if you care that the user isn't constrained by the user interface. If the service is just used to retrieve the user's information via some known credential, and the user can't guess other user's credentials, then who cares if they want to call the service directly? If there isn't anything to hide, everything can just stay in the open.

No matter which, test your service layer. Use a proxy like Zed Attack Proxy to profile the service, and then use a fuzzer to test. If you aren't comfortable testing your own layer, pay a pentester to do it. If you don't test the service layer, I can guarantee that someone else will.

Applying Least Privilege

See to it that your app only has the capabilities it needs. The fewer things that your application touches, the less that an attacker has the potential to use it for bad things. Review the general capabilities and turn everything off you don't need:

- Internet
- Location
- Microphone
- Music Library
- Pictures Library
- Private Networks
- Proximity
- Removable Storage
- Videos Library
- Webcam

And very carefully consider the use of the special capabilities:

- Documents Library
- Enterprise Authentication
- Shared User Credentials

Those are admin level stuff, and should not be needed by 90% of all apps.

In Conclusion

Microsoft has built a pretty secure store ecosystem. They were classic Microsoft, and considered the mistakes of others very carefully before jumping off into their own thing. For the most part, they were very, very successful.

That said, there is little protection for bad code. Building insecure apps will make life easier for the attackers, and using the ecosystem improperly will weaken the whole environment. Taking care to code securely, test and review configuration with the Good Ideas in mind will make for a quality experience for your users and all users.

References

1. Protecting you from Malware, Steven Sinofsky, September 15 2011. http://blogs.msdn.com/b/b8/archive/2011/09/15/protecting-you-from-malware.aspx
2. Security in Windows Store Apps, Josh Dunn, September 2012. http://channel9.msdn.com/Events/Build/2012/3-123
3. Exploit Mitigation Improvements in Windows 8, Ken Johnson and Matt Miller, September 2012. http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf