UNIVERSITY OF PIRAEUS

Department of Digital Systems

Postgraduate Program

Digital Systems Security

Master Course Thesis

Windows Phone 8.1 File Fuzzer

Sotiris Tsiamouris – MTE/1135

Supervisor: Dr. Christos Xenakis

Advisor: Dr. Christoforos Dadoyan

Piraeus, November 2014

Abstract

According to wikipedia fuzz testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems.

The field of fuzzing originates with Barton Miller at the University of Wisconsin in 1988.

"The original work was inspired by being logged on to a modem during a storm with lots of line noise. And the line noise was generating junk characters that seemingly were causing programs to crash. The noise suggested the term 'fuzz'"-Barton Miller.

In this thesis we will introduce a file fuzzing tool called "Windows Phone File Fuzzer" for Windows Phone 8.1. This tool is a "black box"/mutilation-based fuzzing tool for ascii and binary file formats based in the concept of FileFuzz presented by Michael Sutton, Adam Greene and Pedram Amini in *Fuzzing: Brute Force Vulnerability Discovery*.

Table of Contents

Illustration Index

Index of Tables

1. Introduction

Vulnerability discovery is a critical part of security research and software development. Whether you perform a penetration testing, auditing a source code, or developing new software-software vulnerabilities are a critical issue to solve.

Source code auditing is a white box technique used to discover vulnerabilities in software problems. This technique requires the auditor to know every programming concept of the software, the product's operating environment and the source code of the program must be available.[1] The basic problems of source code auditing are, the lack of automation, the need of available source code and the time and effort needed from the auditor to complete the auditing tasks.

The need of a fast and automated source code auditing method brought Fuzzing to the fore. Fuzzing can be implemented without knowing the source code (black-box) and it is fully or semi automated.-saving the auditor from a great workload.

Fuzzing focuses on finding some critical defects quickly, and the found errors are usually very real. Fuzzing can also be performed without any understanding the inner workings of the tested system, whereas code auditing requires full access to source code. Code auditing is usually able to uncover more issues over time, but it also finds more false positives that need to be manually verified by an expert before they can be declared real, critical errors.[2]

Neither fuzzing nor code auditing is able to provably find all possible bugs and defects in a tested system or program. As a rule of thumb, the effectiveness of fuzzing is based on how thoroughly it covers the input space of the tested interface (input space coverage), and how good are the representative malicious and malformed inputs for testing each element or structure within the tested interface definition (quality of generated inputs).[2]

Fuzzing is especially useful in analyzing proprietary systems, as it does not require any access to source code. The system under test can be viewed as a black-box, with one or more external interfaces available for injecting tests, but without any other information available on the internals of the tested system. Having access to information such as source code, design or implementation specifications, debugging or profiling hooks, logging output, or details on the state of the system under test or its operational environment will help in root cause analysis of any problems that are found, but none of this is strictly necessary. Having any of this information available turns the process into gray-box testing, and is recommended for organizations that have access to the details of the systems under test.[2]

Fuzz testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems.[3]

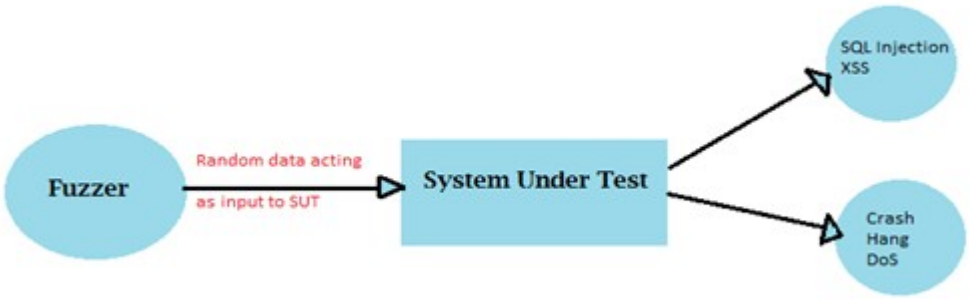


Illustration 1: Fuzzing concept [4]

The scope of fuzzing is to trigger the events in the software's source code that the developer could not or did not handle.

Fuzzing attack types as presented in OWASP:

A fuzzer would try combinations of attacks on:[5]

- numbers (signed/unsigned integers/float...)
- chars (urls, command-line inputs)
- metadata: user-input text (id3 tag)
- pure binary sequences

A common approach to fuzzing is to define lists of "known-to-be-dangerous values" (fuzz vectors) for each type, and to inject them or recombinations:[5]

- for integers: zero, possibly negative or very big numbers
- for chars: escaped, interpretable characters / instructions (ex: For SQL Requests, quotes / commands...)
- for binary: random ones

1.1 Fuzzing Categorization by Type

Fuzzing can be categorized based on the technique that it use to generate the test cases. The two basic categories are:

1. Mutation based, or Black-Box, or Dumb.
2. Generation based, or White-Box, or Intelligent.

Which will be explained below.

Other fuzzing categories are Brute-Force Fuzzing, Pattern-Matching Fuzzing and Hybrid Fuzzing.

1.1.1 Mutation Based

Mutation based Fuzzing do no need any knowledge of the software to be fuzzed. Mutation based Fuzzers use a sample valid input and they create the mutated test cases based on the instructions the user gave to them.

The advantages of Mutation based Fuzzing is that the fuzzer requires minimal setup and no knowledge of the input format.

The disadvantages of Mutation based Fuzzing are:

- The mutated files might not be able to pass the input validation tests of the software being fuzzed-checksums for example.
- It's a time consuming technique. For example a 20kb Microsoft word file needs 20,480 separate files in order fuzz every byte of the file.

Nevertheless despite of all the disadvantages of Mutation based Fuzzing it is the best way so far for black-box fuzzing and the fact that CERT is using it (FOE), means that it is still getting the work done.

1.1.2 Generation Based

Generation based Fuzzing create new test cases based on the model of the input the software being fuzzed takes.

The advantages of Generation based Fuzzing are:

- A smaller amount of test cases is created.
- The test cases are based on the input format.
- The test cases are passing software validation techniques.

The major disadvantage of Generation based fuzzing is that it needs to know the input format. This makes it hard to create test cases for new and/or not known inputs.

Generation based Fuzzing tools are the most efficient tools for fuzzing. Some of the most used tools are Generation based:

- SPIKE
- Peach
- Radamsa (has mutation based techniques too)
- Sulley

1.2 Fuzzing Categorization by Target

Based on the target (fuzzed) software Fuzzing is further categorized. The three most important categories are presented bellow.

Application Fuzzing:

Whatever the fuzzed system is, the attack vectors are within it's I/O. For a desktop app:

- the UI (testing all the buttons sequences / text inputs)
- the command-line options
- the import/export capabilities (see file format fuzzing below)

For a web app: urls, forms, user-generated content, RPC requests, etc.[5]

Protocol Fuzzing:

A protocol fuzzer sends forged packets to the tested application, or eventually acts as a proxy, modifying requests on the fly and replaying them.[5]

File Fuzzing:

A file format fuzzer generates multiple malformed samples, and opens them sequentially. When the program crashes, debug information is kept for further investigation.

One can attack:

- the parser layer (container layer): file format constraints, structure, conventions, field sizes, flags, etc.
- the codec/application layer: lower-level attacks, aiming at the program's deeper internals.

1.3 Windows Phone

Windows Phone (WP) is a smartphone operating system developed by Microsoft. It is the successor to Windows Mobile, although it is incompatible with the earlier platform. With Windows Phone, Microsoft created a new user interface, featuring a design language named "Modern" (which was formerly known as "Metro"). Unlike its predecessor, it is primarily aimed at the consumer market rather than the enterprise market. It was first launched in October 2010 with Windows Phone 7.[6]

Windows Phone 8.1 is the current generation of Microsoft's Windows Phone mobile operating system, succeeding Windows Phone 8. It was initially introduced at Microsoft's Build Conference in San Francisco, California, on April 2, 2014. It was released in final form to Windows Phone developers on April 14, 2014, reached general availability on July 15, 2014 and is officially supported by Microsoft as of August 3, 2014.[6]

Windows Phone 8.1 major improvements can be spotted on its filesystem-Windows Phone 8.1 allows apps to view all files stored on a WP device and move, copy, paste and share the contents of a device's internal file system-also Windows Phone 8.1 adds support for closing apps by swiping down on them in the multitasking view (invoked by doing a long-press on the "back" button), which is similar to how multitasking operates on Windows 8 and iOS. Pressing the back button now suspends an app in the multitasking view instead of closing it.[6]

Apps for Windows Phone 8.1 can now be created using the same application model as Store apps for Windows 8.1, based on the Windows Runtime, and the file extension for WP apps is now ".appx" (which is used for Windows Store apps), instead of Windows Phone's traditional ".xap" file format. Applications built for WP8.1 can invoke semantic zoom, as well as access to single sign-on with a Microsoft account. The Windows Phone Store now also updates apps automatically. The store can be manually checked for updates available for applications on a device. It also adds the option to update applications when on Wi-Fi only.[6]

App developers will be able to develop apps using C# / Visual Basic.NET (.NET), C++ (CX) or HTML5/JavaScript, like for Windows 8.[6]

Developers will also be able to build "universal apps" for both Windows Phone 8.1 and Windows 8 that share almost all code, except for that specific to the platform, such as user interface and phone APIs.[6]

Any universal apps that have been installed on Windows 8.1 will automatically appear in the user's "My Apps" section on Windows Phone 8.1.[6]

Apps built for Windows Phone 8 and Windows Phone 7 automatically run on Windows Phone 8.1, but apps built for Windows Phone 8.1 will not run on any previous version of Windows Phone.[6]

1.3.1 Windows Phone Architecture

Windows Phone 8.1 is based on the Windows NT kernel. Windows Phone 8.1 uses the Core System from Windows, which is a minimal Windows system that boots, manages hardware and resources, authenticates and communicates on a network, and contains low-level security features. To handle phone-specific tasks, the Core System is supplemented by a set of Windows Phone specific binaries that form the Mobile Core.[7]

The following illustration provides a high-level overview of the operating system, organized by layer and ownership. Partners provide most of the low-level hardware interaction and boot drivers for the phone, in the form of a board support package (BSP). The BSP is a collection of drivers and support libraries. The core of the BSP is written by the silicon vendor that creates the CPU. OEMs are responsible for adding the drivers required to support the phone hardware, some of which they write themselves, and some of which are provided by IHVs that build specific hardware components. Meanwhile, the kernel and OS largely come from Windows with Windows Phone specific modifications. On the layer above the kernel are the system services and programming frameworks that applications use to create the user experiences of the phone.[7]

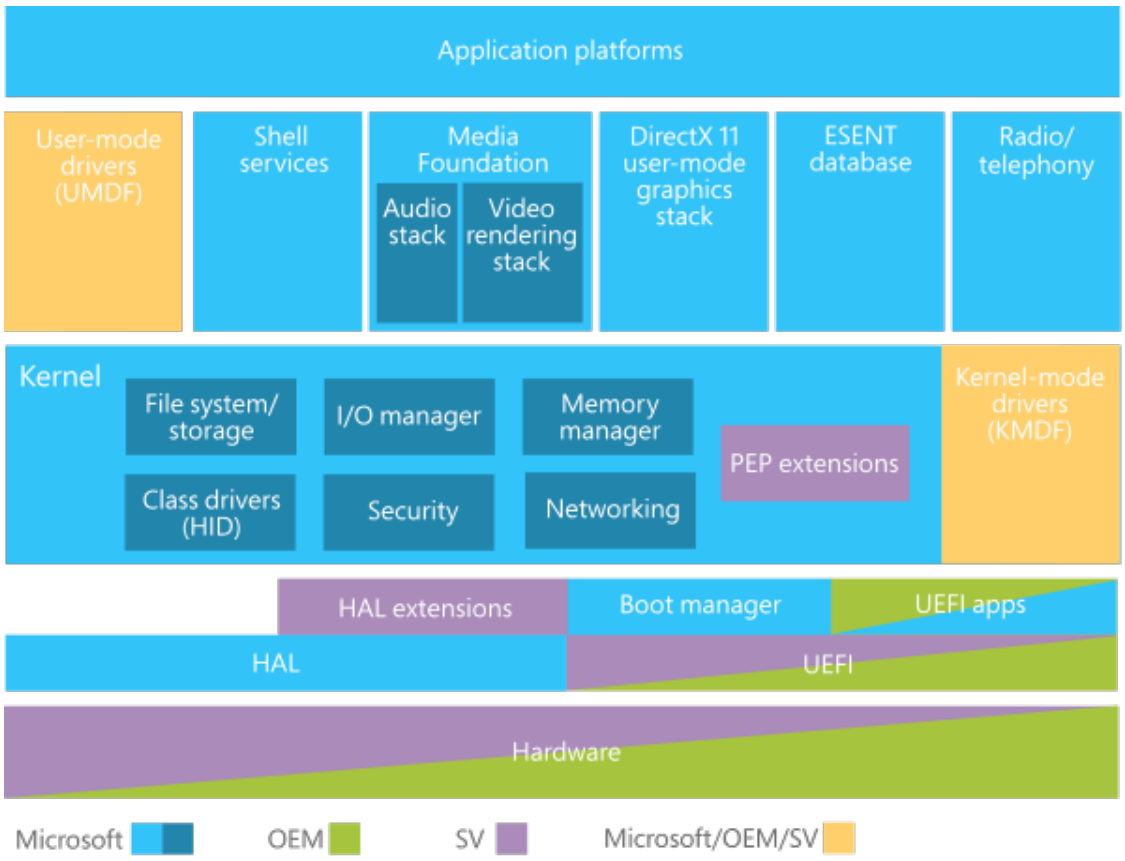


Illustration 2: Windows Phone Architecture[7]

1.3.2 Windows Phone Runtime

Windows Runtime, or WinRT, is a platform-homogeneous application architecture on the Windows 8 operating system. WinRT supports development in C++/CX (Component Extensions, a language based on C++) and the managed languages C# and VB.NET, as well as JavaScript and TypeScript. WinRT applications natively support both the x86 and ARM architectures, and also run inside a sandboxed environment to allow for greater security and stability. WinRT components are designed with interoperability between multiple languages and APIs in mind, including native, managed and scripting languages.[8]

Windows Phone 8 uses a version of the Windows Runtime known as the Windows Phone Runtime. It enables app development in C#, VB.NET and development of Windows Runtime components in C++/CX.[8]

Windows Runtime support on Windows Phone 8.1 converges with Windows 8.1. The release brings a complete Windows Runtime API to the platform, including Windows Runtime XAML support, as well as language bindings for C++/CX and HTML5/JavaScript. There is also a project type called *Universal apps* that enables apps to share code across Windows Phone 8.1 and Windows 8.1 platforms.[8]

Windows Phone Runtime uses the AppX package format from Windows 8, after previously been using Silverlight XAP.[8]

Almost everything in WinRT has been ported over to the phone. This includes the entire Windows Runtime XAML stack, file I/O, Live tiles api, background agents, mapping and geofencing apis etc. Those apis that hasn’t been ported over comes in two categories:

1. Apis that doesn’t make sense on the Phone platform; i.e Printing, search contract etc.
2. Apis that we’re dropped due to time constraints. These are few and hopefully they will be converged over time.

A solution for *Universal* app that targets both Windows and Windows Phone contains three parts:

- A Windows 8.1 Project.
- A Windows Phone 8.1 Windows Runtime Project.
- A Shared folder. Everything you put here will be automatically included in both projects.

A WinRT component can be shared between Windows and Windows Phone, so the code can be reused easily between projects.

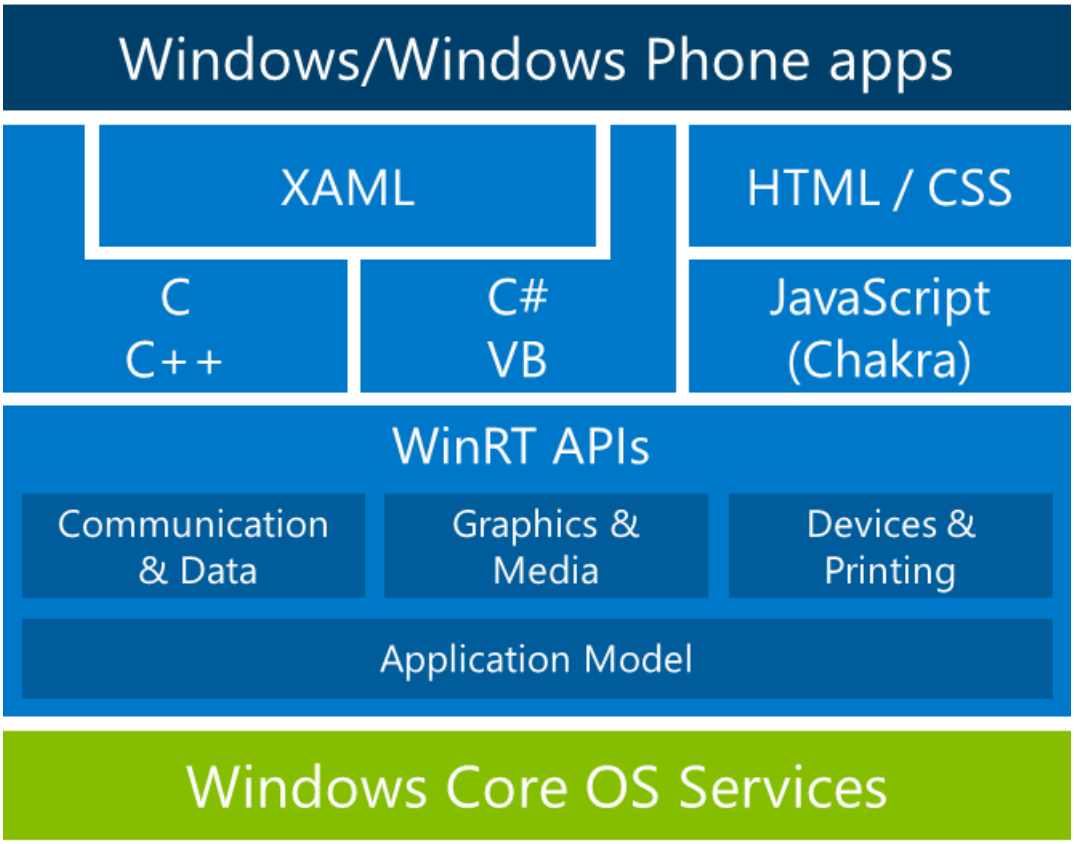


Illustration 3: Windows Runtime Execution Model

1.4 Windows Phone 8.1 Security

Microsoft made some major improvements in its new release of Windows Phone-Windows Phone 8.1. This section focuses in Windows Phone 8.1 system and application security, in order to make clear the difficulty of File Fuzzing.

Initially Windows Phone 8.1 uses Trusted Boot, UEFI Secure Boot verifies that the boot loader is trusted, and then Trusted Boot protects the rest of the startup process by verifying that all Windows boot components have integrity and can be trusted. The boot loader verifies the digital signature of the Windows Phone kernel before loading it. The Windows Phone kernel, in turn, verifies every other component of the Windows startup process, including the boot drivers and startup files.[9]

If a file has been modified (for example, if malware has modified the file to launch malicious code), Trusted Boot protects all of the Windows components and prevents any components that have been tampered with from starting.[9]

After Trusted Boot has completed the startup process, Windows Phone loads the system components and any apps that are loaded automatically at startup. The system components and apps must be properly signed before Windows Phone will load and start them. If a malicious user or code has tampered with the system component or app files, the corresponding component or app will not

be loaded and started.[9]

Unsigned apps are unable to run on Windows Phone, because an app must be signed to be in the Windows Store or be signed with the organization's enterprise development signature. Because all system components and apps must be signed, it is extremely difficult for attackers to run malicious code on a device.[9]

1.4.1 Apps Security

Securing the Windows Phone operating system core is the first step in providing a defense-in-depth approach to securing Windows Phone devices. Securing the apps running on the device is equally important, because attackers could potentially use apps to compromise Windows Phone operating system security and the confidentiality of the information stored on the device.[9]

Windows Phone can mitigate these risks by providing a secured and controlled mechanism for users to acquire trustworthy apps. In addition, the Windows Phone Store app architecture isolates (or sandboxes) one app from another, preventing a malicious app from affecting another app running on the device. Also, the Windows Phone Store app architecture prevents apps from directly accessing critical operating system resources, which helps prevent the installation of malware on devices.[9]

Windows Phone Store:

Downloading and using apps published in the Windows Phone Store dramatically reduce the likelihood that a user can download an app that contains malware. All Windows Phone Store apps go through a careful screening process and scanning for malware and viruses before being made available in the store. The certification process checks Windows Phone Store apps for inappropriate content, store policies, and security issues. Finally, all apps must be signed during the certification process before they can be installed and run on Windows Phone devices. In the event that a malicious app makes it's way through the process and is later detected, the Windows Phone Store can revoked access to the app on any devices that have installed it.[9]

In the end, the Windows Store app-distribution process and the app sandboxing capabilities of Windows Phone 8.1 will dramatically reduce the likelihood that users will encounter malicious apps on the system.[9]

AppContainer:

The Windows Phone security model is based on the principle of least privilege and uses isolation to achieve it. Every app and even large portions of the operating system itself run inside their own isolated sandbox called an AppContainer.[9]

An AppContainer is a secured isolation boundary that an app and its process can run within. Each AppContainer is defined and implemented using a security policy. The security policy of a specific AppContainer defines the operating system capabilities to which the processes have access within the AppContainer. A capability is a Windows Phone device resource such as geographical location information, camera, microphone, networking, or sensors.[9]

By default, a basic set of permissions is granted to all AppContainers, including access its own isolated storage location. In addition, access to other capabilities can be declared within the app code itself. Access to additional capabilities and privileges cannot be requested at runtime, as can be done with traditional desktop applications.[9]

The AppContainer concept is advantageous for the following reasons:

- Attack surface reduction. Apps get access only to capabilities that are declared in the application code and are needed to perform their functions.[9]
- User consent and control. Capabilities that apps use are automatically published to the app details page in the Windows Phone Store. Access to capabilities that may expose sensitive information, such as geographic location, automatically prompt the user to acknowledge and provide consent.[9]
- Isolation. Unlike desktop style apps, which have unlimited access to other apps, communication between Windows Phone apps is tightly controlled. Apps are isolated from one another and can only communicate using predefined communications channels and data types.[9]

Like the Windows Store security model, all Windows Store apps follow the security principal of least privilege. Apps receive the minimal privileges they need to perform their legitimate tasks only, so even if an attacker exploits an app, the damage the exploit can do is severely limited and should be contained within the sandbox. The Windows Phone Store displays the exact permissions that the app requires along with the app's age rating and publisher.[9]

Operating System App Protection:

Windows Phone includes core improvements to make it difficult for malware to perform buffer overflow, heap spraying, and other low-level attacks. For example, Windows Phone includes ASLR and DEP, which dramatically reduce the likelihood that newly discovered vulnerabilities will result in a successful exploit. Technologies like ASLR and DEP act as another level in the defense-in-depth strategy for Window Phone.[9]

- Address space layout randomization. One of the most common techniques for gaining access to a system is to find a vulnerability in a privileged process that is already running, or guess or find a location in memory where important system code and data have been placed, and then overwrite that information with a malicious payload. In the early days of operating systems, any malware that could write directly to system memory could pull off such an exploit: The malware would simply overwrite system memory within well-known and predictable locations. Because all Windows Phone Store apps run in an AppContainer and with fewest necessary privileges, most apps are unable to perform this type of attack outside of one app. It is conceivable that an app from the Window Phone Store might be malicious, but the AppContainer severely limits any damage that the malicious app might do, as apps

are also unable to access critical operating system components. The level of protection AppContainers provide is one of the reasons that their functionality was brought into Windows 8.1 client operating systems. However, ASLR provides an additional defense in-depth to help further secure apps and the core operating system.[9]

- Data execution prevention. Malware depends on its ability to put a malicious payload into memory with the hope that it will be executed later. ASLR makes that much more difficult, but wouldn't it be great if Windows Phone could prevent that malware from running if it writes to an area that has been allocated solely for the storage of information? DEP does exactly that by substantially reducing the range of memory that malicious code can use for its benefit. DEP uses the eXecute Never (XN) bit on the ARM processors in Windows Phone devices to mark blocks of memory as data that should never be executed as code. Therefore, even if an attacker succeeds in loading the malware code into memory, to the malware code will not execute. DEP is automatically active in Windows Phone because all devices have ARM processors that support the XN bit.[9]

Internet Explorer:

Windows Phone includes Internet Explorer 11 for Windows Phone. Internet Explorer helps to protect the user because it runs in an isolated AppContainer and prevents web apps from accessing the system and other app resources. In addition, Internet Explorer on Windows Phone supports a browser model without plug-ins, so plug-ins that compromise the user experience or perform malicious actions cannot be installed (just like the Windows Store version of Internet Explorer in Windows 8.1).[9]

The SmartScreen URL Reputation filter is also available in Internet Explorer for Windows Phone. This technology blocks or warns users of websites that are known to be malicious or are suspicious. [9]

Internet Explorer on Windows Phone can also use SSL to encrypt communication, just as in other Windows operating systems. This is discussed in more detail in the "Communication encryption" section later in this guide.[9]

1.4.3 Summary

Malware resistance is a cornerstone to a security strategy, and Windows Phone devices are designed from the ground up to mitigate or in some cases even eliminate the potential for the most common malware threats. Windows Phone includes the following malware-resistance features:[9]

- UEFI and Secure Boot. Rest easy that malware cannot be introduced during the boot process (bootkits or rootkits). Only trusted, signed software is started and loaded during the startup process. Coupled with UEFI, your devices will be less vulnerable to bootkits and rootkits.[9]
- Trusted Boot. When the boot process is secure, this feature helps ensure that all operating system components and drivers are unmodified and free of malware. Any components that

have been tampered with will be unable to start on the device.[9]

- System and app integrity. This feature builds on Trusted Boot process security by ensuring that all system software (such as Windows services) and apps on Windows Phone are signed and tamper free.[9]
- Windows Phone Store apps. Regardless of whether you use apps from the Windows Phone Store or custom LOB apps, Windows Phone helps ensure that these apps are secure. All Windows Phone Store apps are scanned for malware prior to being published in the store. Internally developed LOB apps must be signed with the organization's developer certificate, ensuring that no malware can be introduced through unauthorized apps. You can even restrict the apps that are available to users for installation from the Windows Phone Store or completely disable access to the Windows Phone Store, if required by organizational or regulatory agency guidelines.[9]
- AppContainer. All apps, and even some system components, on Windows Phone run in an isolated sandbox called an AppContainer. Apps running within an AppContainer run with least privilege and can only gain access to resources in a predefined, declarative manner. Apps running in the AppContainer have limited to no access to the system, other apps, or data.[9]
- Vulnerability mitigations. Technologies like ASLR and DEP help ensure that malware cannot be injected onto your devices through vulnerabilities that may be discovered later.[9]
- Internet Explorer. Browsers are one of the most targeted and common ways to spread malware. Internet Explorer on Windows Phone helps protect the organization's apps and data from attack by using technologies such as AppContainers and SmartScreen.[9]

1.5 Windows Phone 8.1 File Fuzzer Overview

Windows Phone 8.1 File Fuzzer is a Windows Phone 8.1 application. It is inspired by FileFuzz and its is programmed to be user friendly and easy to use.

The scope of the app is to help security researchers and software developers to test build-in Windows Phone file handling applications such as Windows Office, Picture Viewer, Music Player and any other application that handles files. Also it gives the ability to software developers to test their applications for Buffer Overflows, Memory Leaks, DoS, etc.

The application was developed using C# under Microsoft Visual Studio 2013 Express and tested in a Nokia Lumia 520 device.

1.5.1 Considerations

Anyone interested to develop a file fuzzing application in Windows Phone 8.1 needs to take into consideration the facts bellow:

1. A mobile device for testing is necessary. Windows Phone 8.1 emulator does not let the user to download apps, files, etc.
2. Windows Phone 8.1 takes over after a file is launched. The application which launched the file is suspended and the application that handles the file takes over. The user need to use the back button to get back to the application. That makes launching multiple files at once almost impossible.
3. An OEM account is needed in order to use the attach-to-process Windbg future and Tshell for Windows Phone.
4. Fuzzer should target a basic Windows Phone 8.1 application-most users are using them-but Microsoft might fuzzing them with DFF and definitely run them inside a sandboxed environment(so the amount of vulnerable crashes are limited).

1.5.2 Overview

Windows Phone 8.1 File Fuzzer is a Mutation based Fuzzer developed with C#. It has the ability to Fuzz ascii and binary files and launch them in Windows Phone 8.1.

The basic two components of the application is the Ascii Test Case Creator and the Binary Test Case Creator.

Ascii Test Case Creator is taking the data that the user provides and builds a *N* number of test cases depending on the number of occurrences of the char that the user searched for *C*, or the total number of occurrences of *C* in the template file.

Binary Test Case Creator is taking the data the user provides and builds a *N* number of test cases depending on the length of the binary file, the range of bytes the user wants to overwrite , or the specific byte the user wants to overwrite.

Windows Phone 8.1 File Fuzzer will be analyzed in more detail on the next chapter.

1.6 Summary

Fuzzing is not measured by a standard. There are no good or bad techniques. If the fuzzer works and produces crashes then it is a good fuzzer.

The security researcher first of all should investigate the target and after that choose or build the

proper tool for the work.

A Fuzzing process follows the order below:

1. Choose a target.
2. Investigate the target's inputs.
3. Create the test cases.
4. Launch a test case.
5. Monitor the target for flaws.
6. Kill the test case.
7. Repeat.

Data Generation tips:

1. Smart integers (max32, max16, max8)
2. Smart strings (long strings)
3. Field delimiters (space, tab, new line)
4. Format strings (%s, %n)
5. Character translation (0xFE, 0xFF)
6. Directory Traversal
7. Command Injection

Detection tips (info about the process):

1. Event Logs
2. Debuggers
3. Return Calls
4. Test Case metadata

2. Windows Phone 8.1 File Fuzzer Development

Windows Phone File Fuzzer 8.1 development will be presented in this chapter. Application's design and functionality will be extensively analyzed.

2.1 Windows Phone File Fuzzer Design

Windows Phone 8.1 File Fuzzer has a simple and user-friendly design in order to make the fuzzing process easy for every user with a mid knowledge on the subject.

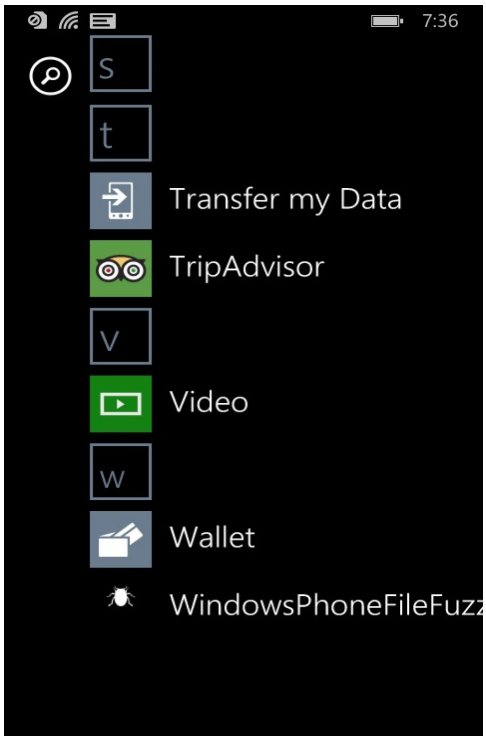


Illustration 4: Windows Phone 8.1 File Fuzzer: Icon

The first page prompts the user to choose what type of file to fuzz.

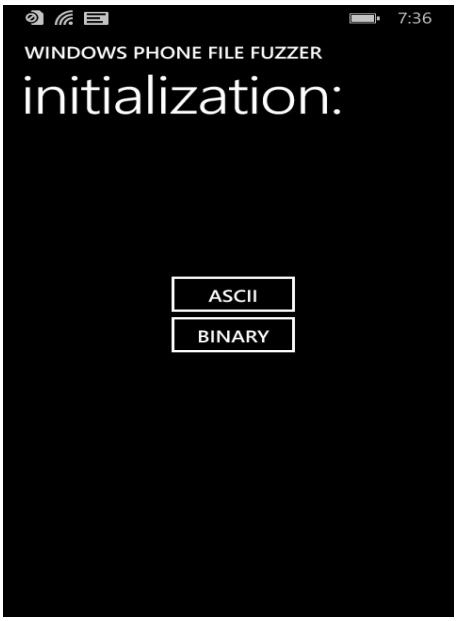


Illustration 5: WP8.1 File Fuzzer: Main Page

After choosing the file type the user is navigated to the corresponding page.

If the user choose the ASCII selection, the app will navigate to the Ascii Page where the user can fill the fuzzing data for an ascii file format.

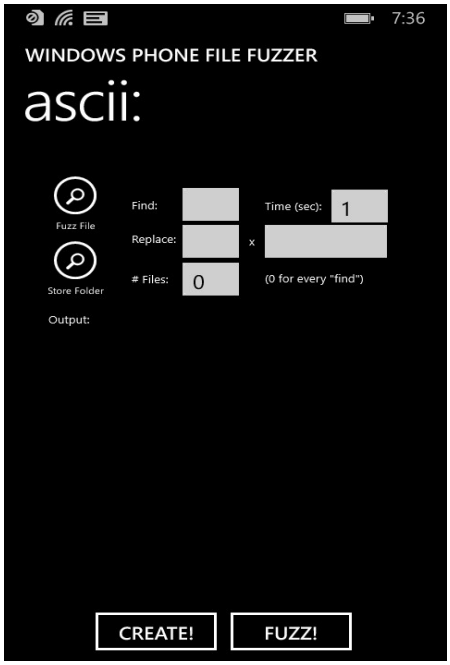


Illustration 6: Windows Phone 8.1 File Fuzzer: Ascii Page

If the user choose the BINARY selection, the app will navigate to the Binary Page where the user can fill the fuzzing data for a binary file format.

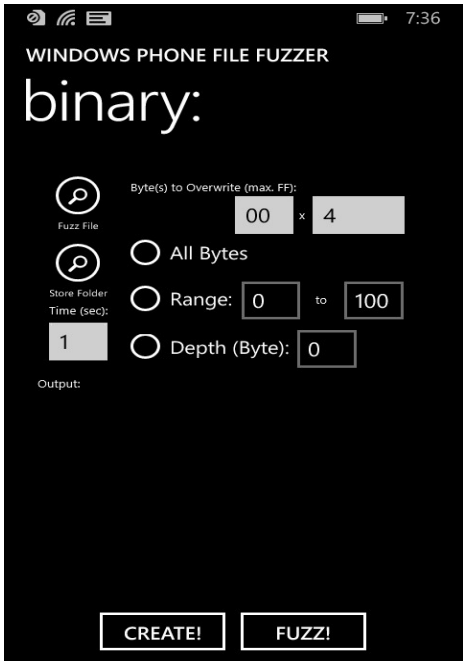


Illustration 7: Windows Phone 8.1 File Fuzzer: Binary Page

2.1.1 Analysis of the Pages

ASCII Page:

In the ascii page the user can add the following data:

1. Fuzz File: The template file to create the test cases.
2. Store Folder: The folder to store the test cases.
3. Time: The time to run every test case when it launch.
4. Find: The char to search for inside the file.
5. Replace: The char to add after the “Find” char.
6. X: the times to multiply the char the user adds.
7. #Files: The number of test cases the user wants to create. If 0 it creates a test case for each “find” char occurrence.
8. CREATE! button: It creates the test cases.

9. FUZZ! Button: Launches the test cases.
10. Output: A text block that informs the user.

BINARY Page:

In the Binary Page the user can add the following data:

1. Fuzz File: The template file to create the test cases.
2. Store Folder: The folder to store the test cases.
3. Time: The time to run every test case when it launch.
4. Byte(s) to Overwrite: The byte the user add to the test case.
5. X: The times to multiply the byte the user adds.
6. All Bytes (Selection): The fuzzer creates a test case for every byte of the file until byte = file_length-user_created_byte_array.
7. Range (Selection): The fuzzer creates a test case for every byte in the range the user added until byte = file_length-user_created_byte_array.
8. Depth (Selection): The fuzzer creates one test case for the byte the user added. The byte = file_length-user_created_byte_array.
9. CREATE! button: It creates the test cases.
10. FUZZ! Button: Launches the test cases.
11. Output: A text block that informs the user.

2.2 Windows Phone 8.1 File Fuzzer Functionality

File and Folder Selection:

In order to select the template file and the store folder, FileOpenPicker and FolderPicker classes from Windows.Storage.Pickers were used.

Parsing:

Parsing was used in order to avoid application's crashes from false data added by the user.

```
public static byte[] StringToByteArray(String hex)
{
    //Convert a "hexstring" to a byte array
    int NumberChars = hex.Length / 2;
    byte[] bytes = new byte[NumberChars];
    using (var sr = new StringReader(hex))
    {
        for (int i = 0; i < NumberChars; i++)
            bytes[i] =
              Convert.ToByte(new string(new char[2] { (char)sr.Read(), (char)sr.Read() }), 16);
    }
    return bytes;
}
```

Illustration 8: Hexstring Parsing Example

Ascii Test Case Creation:

During this process the fuzzer copies the template file to the store folder the user choose and based on the file's name and data the test cases are created. For example if the template's file name is *"test.txt"* the fuzzer will name the test cases as *"test0.txt".....”testn.txt"*.

```
int at = 0;
int start = 0;
int count = 0;
string toFile = "";
string fileContent = await FileIO.ReadTextAsync(newFile);
while ((start < fileContent.Length) && (at > -1) && (count < n))
{
    toFile = fileContent;
    string addString = "";
    at = fileContent.IndexOf(find, start);
    if (at == -1) break;
    addString = toFile.Insert(at + 1, fuzzString);
    StorageFile testCase = await fuzzFolder.CreateFileAsync(originalName+count+ext, CreationCollisionOption.ReplaceExisting);
    await FileIO.WriteTextAsync(testCase, addString);
    start = at + 1;
    count++; //if equal to #Files break
}
```

Illustration 9: Windows Phone 8.1 File Fuzzer: Create Ascii Test Cases

Binary Test Case Creation:

The file names of the binary test cases are created similar to the Ascii Test Case Creation above. The difference is the use of a Stream to alter and store the data.

```
while (start < size - Convert.ToUInt64(m))
{
    StorageFile testCase = await fuzzFile.CopyAsync(fuzzFolder, originalName + count + ext, NameCollisionOption.ReplaceExisting)
    count++;
    StorageFile sampleFile = await fuzzFolder.GetFileAsync(testCase.Name);
    IRandomAccessStream writeStream = await sampleFile.OpenAsync(FileAccessMode.ReadWrite);
    IOutputStream outputStream = writeStream.GetOutputStreamAt(start);
    DataWriter dataWriter = new DataWriter(outputStream);
    dataWriter.WriteBytes(b);
    await dataWriter.StoreAsync();
    await outputStream.FlushAsync();
    sb.Append("File: ").Append(testCase.Name).Append(" created.").Append(Environment.NewLine);
    start++;
}
```

Illustration 10: Windows Phone 8.1 File Fuzzer: Create Binary Test Cases

Launching Test Cases:

The test cases are launched using Windows.System.Launcher.

```
string launchFile = originalName+fileCount+ext;
var file = await fuzzFolder.GetFileAsync(launchFile);

if (file != null)
{
   // sb.Append("File ").Append(file.Name).Append(" launched successfully").Append(Environment.NewLine);
    //fileCount++;
    // Launch the retrieved file

    var success = await Windows.System.Launcher.LaunchFileAsync(file);

    if (success)
    {
        //System.Threading.Thread.Sleep(t)
        sb.Append("File ").Append(file.Name).Append(" launched successfully").Append(Environment.NewLine);
        fileCount++;
    }
```

Illustration 11: Windows Phone 8.1 File Fuzzer: Launch Test Cases

Of course the issue we discussed in section 1.3.1 makes launching a serious problem.

3. Windows Phone 8.1 File Fuzzer Demonstration

In this chapter we will demonstrate the use of the application. Also we will see the files the fuzzer creates according to the data the user adds.

3.1 Creating Ascii Test Cases

In this section we will see what test cases the fuzzer creates in a saved from the web *facebook.htm* file. The reason to fuzz a *.htm* file is to see how Internet explorer will react openning a malformed file.

The illustration below shows us the creation of the test cases, focusing in char “.

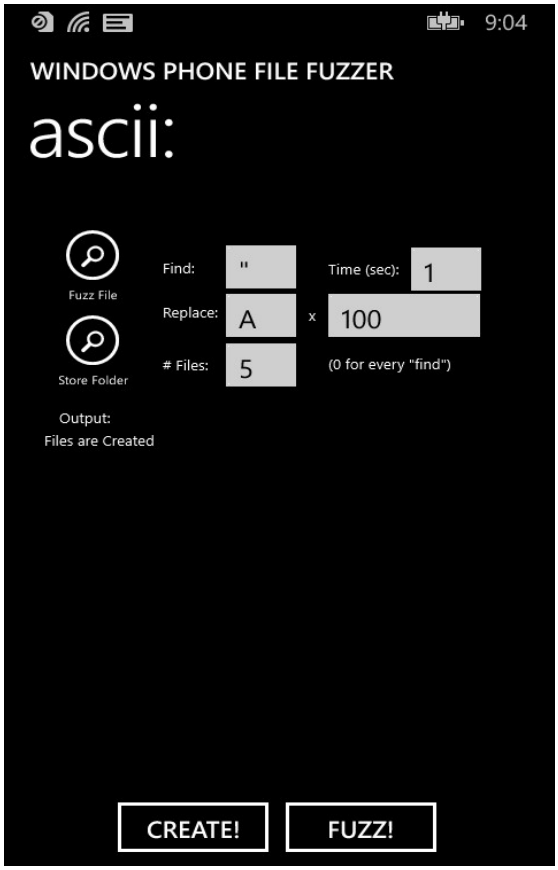


Illustration 12: Windows Phone 8.1 File Fuzzer: Ascii Test Cases Create

The original file was:

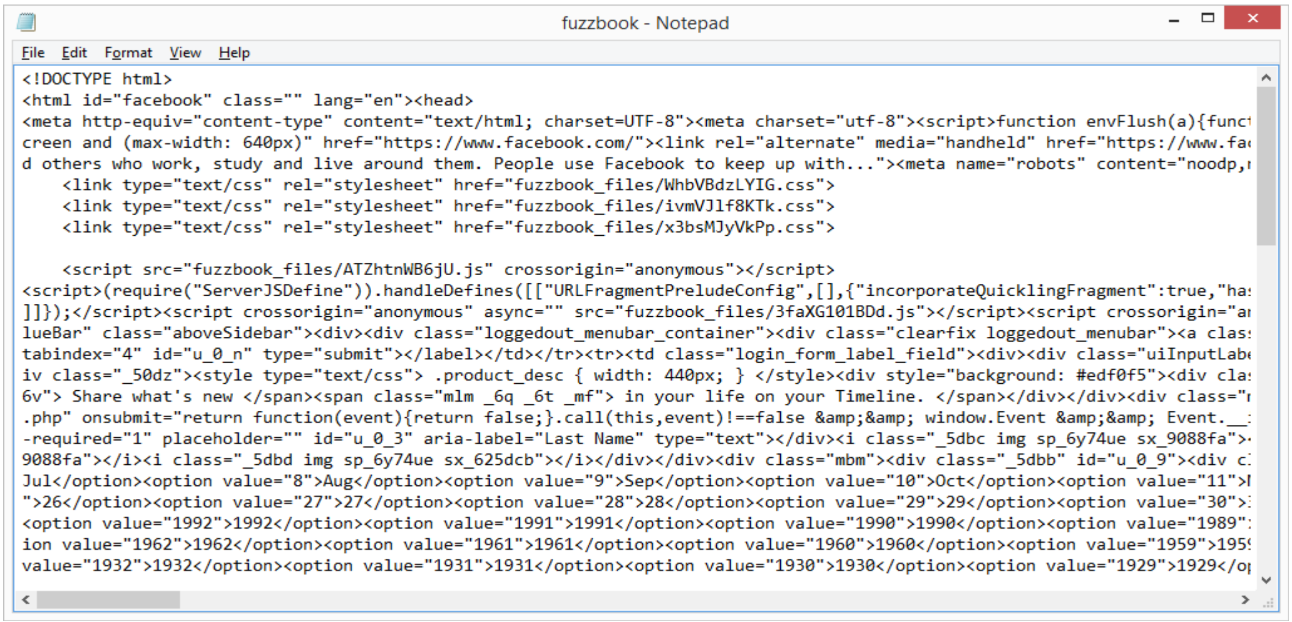


Illustration 13: Facebook.htm

The fuzzed file (test case 1) is:

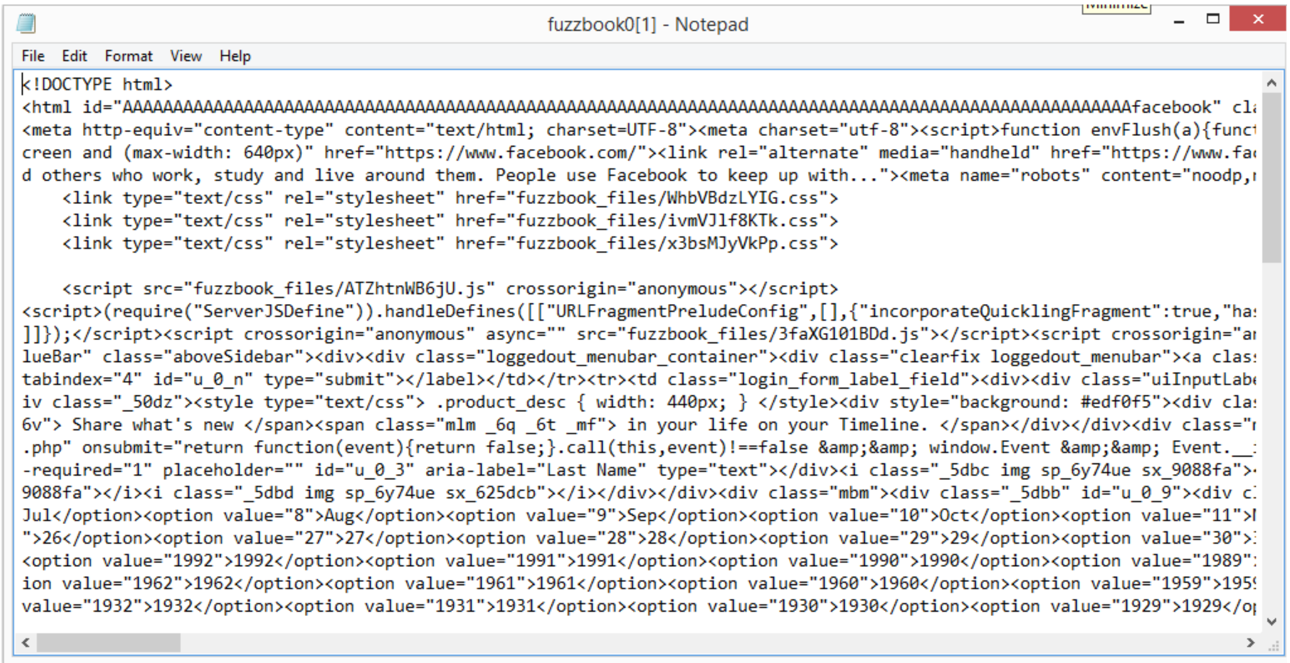


Illustration 14: fuzzbook.htm

3.2 Creating Binary Test Cases

In this section we will see what test cases the fuzzer creates for a .png image.

Creating the test cases:

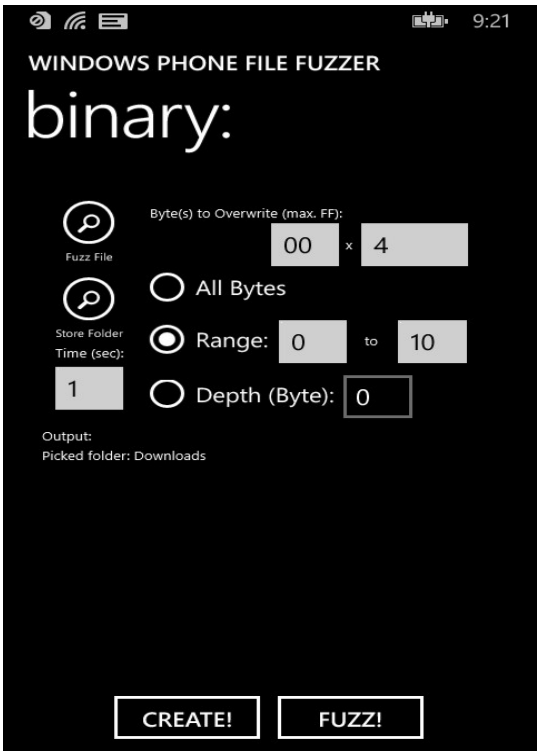
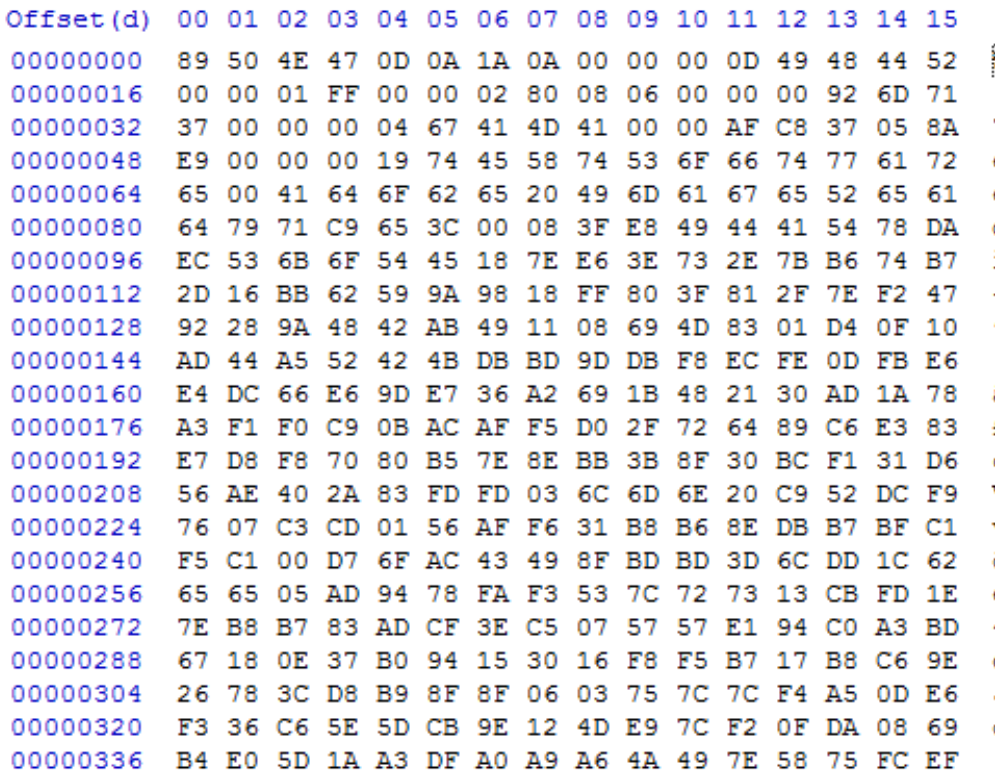


Illustration 15: Windows Phone 8.1 File Fuzzer: Binary Test Cases Create

The bytes of the original file using a decimal offset representation in the hex viewer:

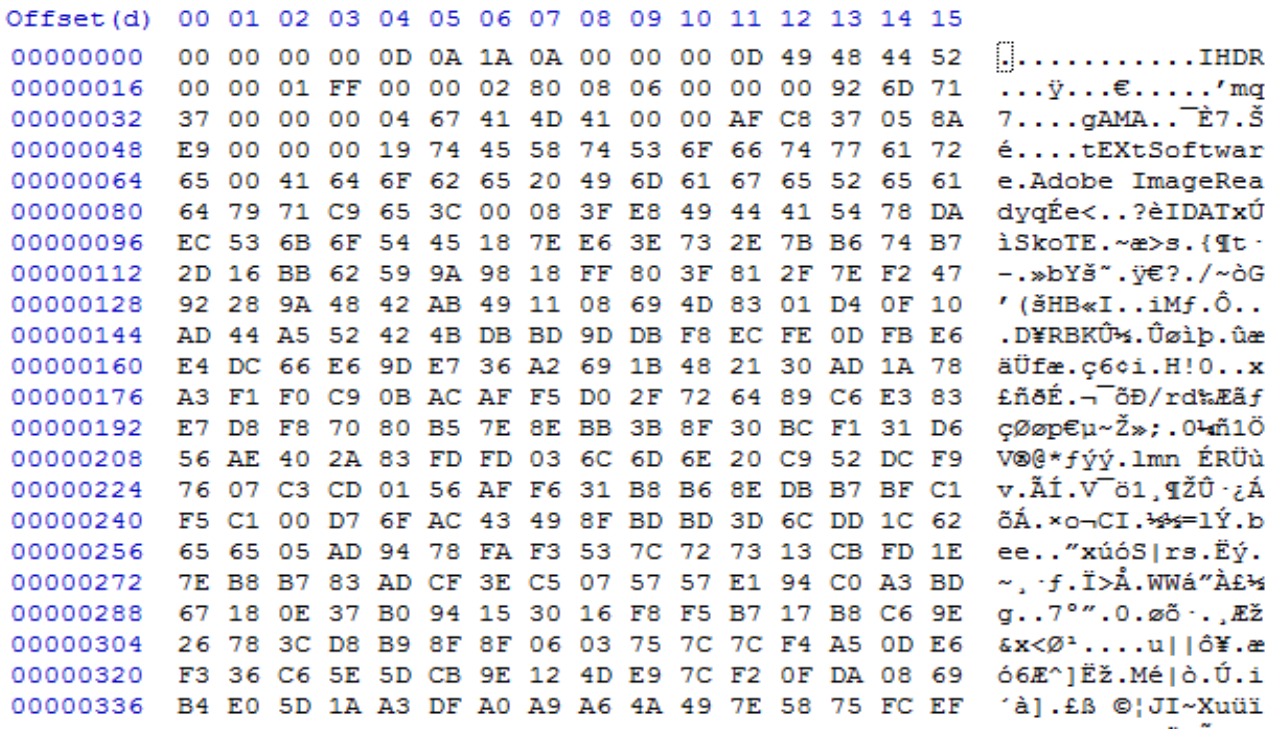


```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15

00000000   89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  ‰PNG........IHDR
00000016   00 00 01 FF 00 00 02 80 08 06 00 00 00 92 6D 71  ...ÿ...€.....'mq
00000032   37 00 00 00 04 67 41 4D 41 00 00 AF C8 37 05 8A  7....gAMA..¯È7.Š
00000048   E9 00 00 00 19 74 45 58 74 53 6F 66 74 77 61 72  é....tEXtSoftwar
00000064   65 00 41 64 6F 62 65 20 49 6D 61 67 65 52 65 61  e.Adobe ImageRea
00000080   64 79 71 C9 65 3C 00 08 3F E8 49 44 41 54 78 DA  dyqÉe<..?èIDATxÚ
00000096   EC 53 6B 6F 54 45 18 7E E6 3E 73 2E 7B B6 74 B7  ìSkoTE.~æ>s.{¶t·
00000112   2D 16 BB 62 59 9A 98 18 FF 80 3F 81 2F 7E F2 47  -.»bYš˜.ÿ€?./~òG
00000128   92 28 9A 48 42 AB 49 11 08 69 4D 83 01 D4 0F 10  ’(šHB«I..iMƒ.Ô..
00000144   AD 44 A5 52 42 4B DB BD 9D DB F8 EC FE 0D FB E6  .D¥RBKÛ½.Ûøìþ.ûæ
00000160   E4 DC 66 E6 9D E7 36 A2 69 1B 48 21 30 AD 1A 78  äÜfæ.ç6¢i.H!0..x
00000176   A3 F1 F0 C9 0B AC AF F5 D0 2F 72 64 89 C6 E3 83  £ñðÉ.¬¯õÐ/rd‰Æãƒ
00000192   E7 D8 F8 70 80 B5 7E 8E BB 3B 8F 30 BC F1 31 D6  çØøp€µ~Ž»;.0¼ñ1Ö
00000208   56 AE 40 2A 83 FD FD 03 6C 6D 6E 20 C9 52 DC F9  V®@*ƒýý.lmn ÉRÜù
00000224   76 07 C3 CD 01 56 AF F6 31 B8 B6 8E DB B7 BF C1  v.ÃÍ.V¯ö1¸¶ŽÛ·¿Á
00000240   F5 C1 00 D7 6F AC 43 49 8F BD BD 3D 6C DD 1C 62  õÁ.×o¬CI.½½=lÝ.b
00000256   65 65 05 AD 94 78 FA F3 53 7C 72 73 13 CB FD 1E  ee..”xúóS|rs.Ëý.
00000272   7E B8 B7 83 AD CF 3E C5 07 57 57 E1 94 C0 A3 BD  ~¸·ƒ.Ï>Å.WWá”À£½
00000288   67 18 0E 37 B0 94 15 30 16 F8 F5 B7 17 B8 C6 9E  g..7°”.0.øõ·.¸Æž
00000304   26 78 3C D8 B9 8F 8F 06 03 75 7C 7C F4 A5 0D E6  &x<Ø¹....u||ô¥.æ
00000320   F3 36 C6 5E 5D CB 9E 12 4D E9 7C F2 0F DA 08 69  ó6Æ^]Ëž.Mé|ò.Ú.i
00000336   B4 E0 5D 1A A3 DF A0 A9 A6 4A 49 7E 58 75 FC EF  ´à].£ß ©¦JI~Xuüï
```

Illustration 16: The Original Binary File

And the first (0) and last (9) fuzzed files:



```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15

00000000   00 00 00 00 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  ............IHDR
00000016   00 00 01 FF 00 00 02 80 08 06 00 00 00 92 6D 71  ...ÿ...€.....'mq
00000032   37 00 00 00 04 67 41 4D 41 00 00 AF C8 37 05 8A  7....gAMA..¯È7.Š
00000048   E9 00 00 00 19 74 45 58 74 53 6F 66 74 77 61 72  é....tEXtSoftwar
00000064   65 00 41 64 6F 62 65 20 49 6D 61 67 65 52 65 61  e.Adobe ImageRea
00000080   64 79 71 C9 65 3C 00 08 3F E8 49 44 41 54 78 DA  dyqÉe<..?èIDATxÚ
00000096   EC 53 6B 6F 54 45 18 7E E6 3E 73 2E 7B B6 74 B7  ìSkoTE.~æ>s.{¶t·
00000112   2D 16 BB 62 59 9A 98 18 FF 80 3F 81 2F 7E F2 47  -.»bYš˜.ÿ€?./~òG
00000128   92 28 9A 48 42 AB 49 11 08 69 4D 83 01 D4 0F 10  ’(šHB«I..iMƒ.Ô..
00000144   AD 44 A5 52 42 4B DB BD 9D DB F8 EC FE 0D FB E6  .D¥RBKÛ½.Ûøìþ.ûæ
00000160   E4 DC 66 E6 9D E7 36 A2 69 1B 48 21 30 AD 1A 78  äÜfæ.ç6¢i.H!0..x
00000176   A3 F1 F0 C9 0B AC AF F5 D0 2F 72 64 89 C6 E3 83  £ñðÉ.¬¯õÐ/rd‰Æãƒ
00000192   E7 D8 F8 70 80 B5 7E 8E BB 3B 8F 30 BC F1 31 D6  çØøp€µ~Ž»;.0¼ñ1Ö
00000208   56 AE 40 2A 83 FD FD 03 6C 6D 6E 20 C9 52 DC F9  V®@*ƒýý.lmn ÉRÜù
00000224   76 07 C3 CD 01 56 AF F6 31 B8 B6 8E DB B7 BF C1  v.ÃÍ.V¯ö1¸¶ŽÛ·¿Á
00000240   F5 C1 00 D7 6F AC 43 49 8F BD BD 3D 6C DD 1C 62  õÁ.×o¬CI.½½=lÝ.b
00000256   65 65 05 AD 94 78 FA F3 53 7C 72 73 13 CB FD 1E  ee..”xúóS|rs.Ëý.
00000272   7E B8 B7 83 AD CF 3E C5 07 57 57 E1 94 C0 A3 BD  ~¸·ƒ.Ï>Å.WWá”À£½
00000288   67 18 0E 37 B0 94 15 30 16 F8 F5 B7 17 B8 C6 9E  g..7°”.0.øõ·.¸Æž
00000304   26 78 3C D8 B9 8F 8F 06 03 75 7C 7C F4 A5 0D E6  &x<Ø¹....u||ô¥.æ
00000320   F3 36 C6 5E 5D CB 9E 12 4D E9 7C F2 0F DA 08 69  ó6Æ^]Ëž.Mé|ò.Ú.i
00000336   B4 E0 5D 1A A3 DF A0 A9 A6 4A 49 7E 58 75 FC EF  ´à].£ß ©¦JI~Xuüï
```

Illustration 17: Fuzzed Binary 0

```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15

00000000   89 50 4E 47 0D 0A 1A 0A 00 00 00 00 00 48 44 52  ‰PNG.........HDR
00000016   00 00 01 FF 00 00 02 80 08 06 00 00 00 92 6D 71  ...ÿ...€.....'mq
00000032   37 00 00 00 04 67 41 4D 41 00 00 AF C8 37 05 8A  7....gAMA..¯È7.Š
00000048   E9 00 00 00 19 74 45 58 74 53 6F 66 74 77 61 72  é....tEXtSoftwar
00000064   65 00 41 64 6F 62 65 20 49 6D 61 67 65 52 65 61  e.Adobe ImageRea
00000080   64 79 71 C9 65 3C 00 08 3F E8 49 44 41 54 78 DA  dyqÉe<..?èIDATxÚ
00000096   EC 53 6B 6F 54 45 18 7E E6 3E 73 2E 7B B6 74 B7  ìSkoTE.~æ>s.{¶t·
00000112   2D 16 BB 62 59 9A 98 18 FF 80 3F 81 2F 7E F2 47  -.»bYš˜.ÿ€?./~òG
00000128   92 28 9A 48 42 AB 49 11 08 69 4D 83 01 D4 0F 10  ’(šHB«I..iMƒ.Ô..
00000144   AD 44 A5 52 42 4B DB BD 9D DB F8 EC FE 0D FB E6  .D¥RBKÛ½.Ûøìþ.ûæ
00000160   E4 DC 66 E6 9D E7 36 A2 69 1B 48 21 30 AD 1A 78  äÜfæ.ç6¢i.H!0..x
00000176   A3 F1 F0 C9 0B AC AF F5 D0 2F 72 64 89 C6 E3 83  £ñðÉ.¬¯õÐ/rd‰Æãƒ
00000192   E7 D8 F8 70 80 B5 7E 8E BB 3B 8F 30 BC F1 31 D6  çØøp€µ~Ž»;.0¼ñ1Ö
00000208   56 AE 40 2A 83 FD FD 03 6C 6D 6E 20 C9 52 DC F9  V®@*ƒýý.lmn ÉRÜù
00000224   76 07 C3 CD 01 56 AF F6 31 B8 B6 8E DB B7 BF C1  v.ÃÍ.V¯ö1¸¶ŽÛ·¿Á
00000240   F5 C1 00 D7 6F AC 43 49 8F BD BD 3D 6C DD 1C 62  õÁ.×o¬CI.½½=lÝ.b
00000256   65 65 05 AD 94 78 FA F3 53 7C 72 73 13 CB FD 1E  ee..”xúóS|rs.Ëý.
00000272   7E B8 B7 83 AD CF 3E C5 07 57 57 E1 94 C0 A3 BD  ~¸·ƒ.Ï>Å.WWá”À£½
00000288   67 18 0E 37 B0 94 15 30 16 F8 F5 B7 17 B8 C6 9E  g..7°”.0.øõ·.¸Æž
00000304   26 78 3C D8 B9 8F 8F 06 03 75 7C 7C F4 A5 0D E6  &x<Ø¹....u||ô¥.æ
00000320   F3 36 C6 5E 5D CB 9E 12 4D E9 7C F2 0F DA 08 69  ó6Æ^]Ëž.Mé|ò.Ú.i
00000336   B4 E0 5D 1A A3 DF A0 A9 A6 4A 49 7E 58 75 FC EF  ´à].£ß ©¦JI~Xuüï
```

Illustration 19: Fuzzed Binary 9

4. Testing

In this chapter the tool will be evaluated (partially-since Microsoft only provides OEM tools to its hardware partners and the critical phase of debugging could not be done). Test cases will be run from the targeted apps and the expected crashes will be recorded and evaluated.

During this stage a great amount of test cases will be executed via various file reading apps of Windows Phone 8.1. A part of these apps will be the standard file reading apps that Microsoft develops/provides through its Windows Phones. An other part will be third party applications developed by non-Microsoft developers but evaluated by Microsoft and available from the Microsoft Store.

The table below presents the files and the file reading apps used for this thesis.

File Type	Application	Description	Fuzzing Type
.png	• Image Viwer • Adobe Photoshop Express • Lomogram+ • Lumia Creative Studio	Portable Network Graphics (PNG), is a raster graphics file format that supports lossless data compression. PNG was created as an improved, non-patented replacement for Graphics Interchange Format (GIF), and is the most used lossless image compression format on the Internet.*	Binary
.jpg	• Image Viwer • Adobe Photoshop Express • Lomogram+ • Lumia Creative Studio	JPEG (seen most often with the .jpg or .jpeg filename extension) is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography. The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality.*	Binary
.bmp	• Image Viwer • Adobe Photoshop Express • Lomogram+ • Lumia Creative Studio	The BMP file format, also known as bitmap image file or device independent bitmap (DIB) file format or simply a bitmap, is a raster graphics image file format used to store bitmap digital images, independently of the display device (such as a graphics adapter), especially on Microsoft Windows and OS/2	Binary

		operating systems.*	
.gif	• Image Viwer • Adobe Photoshop Express • Lomogram+ • Lumia Creative Studio	The Graphics Interchange Format (better known by its acronym GIF; is a bitmap image format that was introduced by CompuServe in 1987 and has since come into widespread usage on the World Wide Web due to its wide support and portability.*	Binary
.zip	• Archiver	ZIP is an archive file format that supports lossless data compression. A ZIP file may contain one or more files or folders that may have been compressed. The .ZIP file format permits a number of compression algorithms.*	Binary
.doc	• Office	DOC or doc (an abbreviation of 'document') is a filename extension for word processing documents, most commonly in the proprietary *Microsoft Word Binary File Format*.*	Binary
.xls	• Office	Microsoft Excel up until 2007 version used a proprietary binary file format called Excel Binary File Format (.XLS) as its primary format.*	Binary
.pdf	• Adobe Reader • Pdf Reader	Portable Document Format (PDF) is a file format used to present	Binary

		documents in a manner independent of application software, hardware, and operating systems. Each PDF file encapsulates a complete description of a fixed-layout flat document, including the text, fonts, graphics, and other information needed to display it.*	
.epub	• Epub Reader	EPUB (short for *electronic publication*) is a free and open e-book standard by the International Digital Publishing Forum (IDPF). Files have the extension *.epub*.*	Binary
.html	• Internet Explorer	A web browser can read HTML files and compose them into visible or audible web pages. The browser does not display the HTML tags, but uses them to interpret the content of the page. HTML describes the structure of a website semantically along with cues for presentation, making it a markup language rather than a programming language.*	Ascii
*From Wikipedia.			

Table 1: File Types

4.1 Ascii Test Cases

The only application tested via Windows Phone 8.1 File Fuzzer using the Ascii test cases creator was Internet Explorer. A html saved page from the web (yahoo greece home page) was mutated for various string occurrences, searching for buffer overflows. Internet Explorer managed to open without crashes all the test cases but during the lack of automation a great amount of possible test cases were not created.

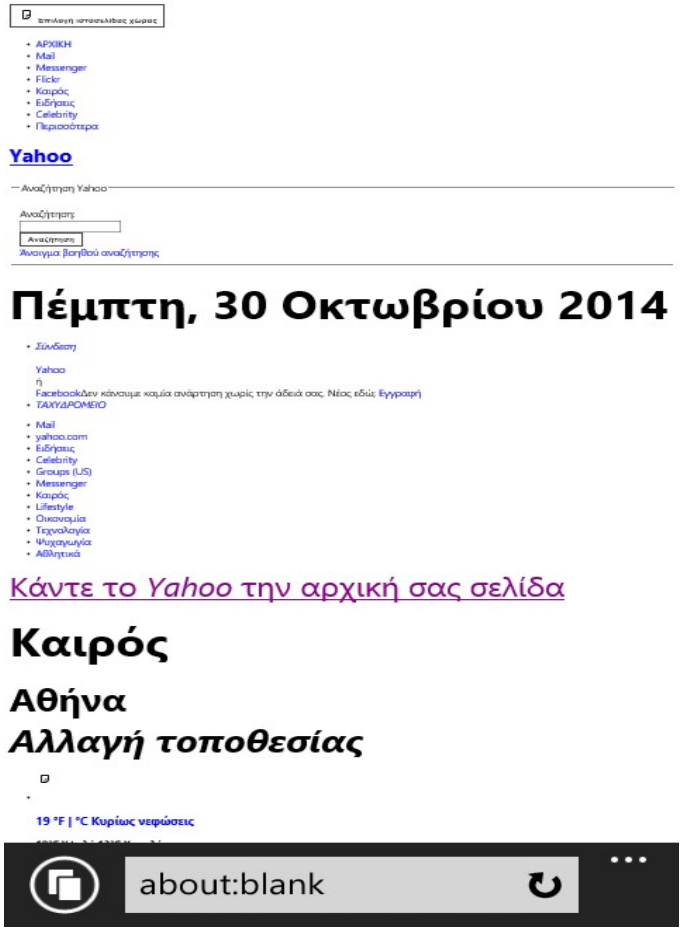


Illustration 20: Internet Explorer opens mutated file

The pictures below shows the mutated html files and the Ascii folder the files are stored.

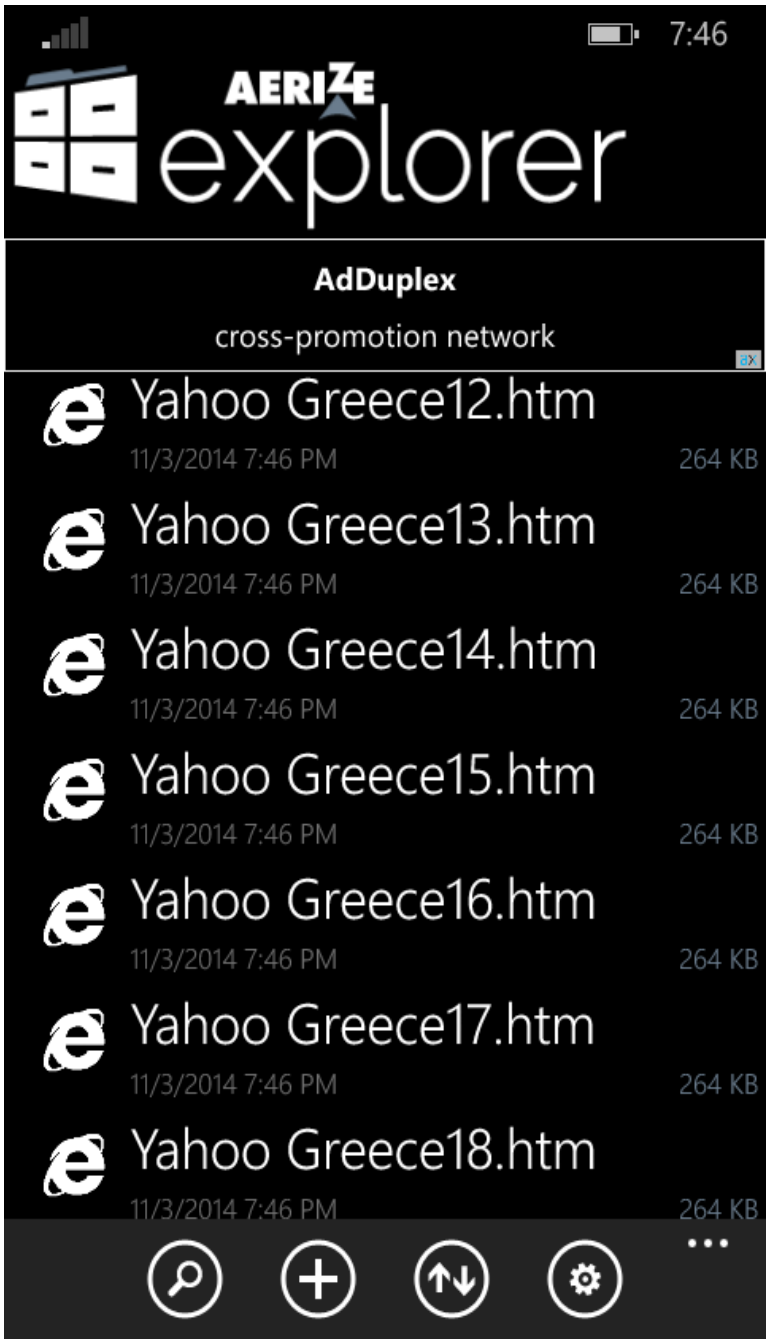


Illustration 21: html file mutations

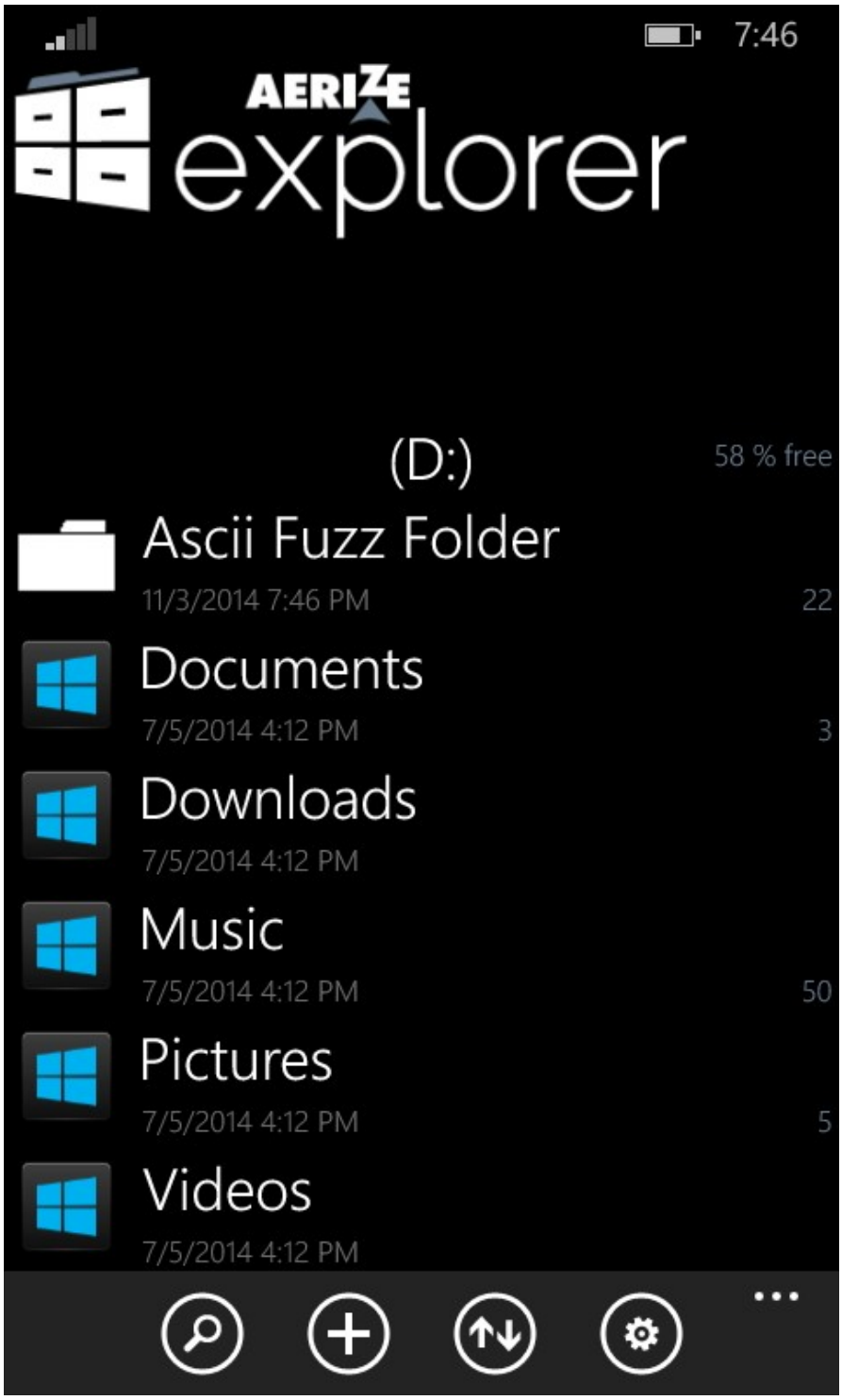


Illustration 22: Ascii Fuzzed Files Folder

4.2 Binary Test Cases

On the contrary a greater amount of files were mutated and tested via the Binary test cases creator. During this process only a suspicion of a crash occurred while .bmp files were launched, but it probably was because of some kind of overload and when the same files were opened did not provided any crashes.

Windows Phone applications managed to open or handle the errors of the files but the lack of automation and the fact that Microsoft din not provided OEM tools is impossible to fully evaluate the tool.

Binary File reading applications were tested for memory leaks and buffer overflows using bytes like MAX_INT, MIN_INT, Ax1000 (for example, looking for buffer overflows), escape characters and magic numbers.

During this process a large amount of test cases were created. For example a test case for each byte of a small image file with approximately 700 KB of file size , creates 700K files, which makes the fuzzing process impossible.

4.2.1 PNG

All the image reading applications (mentioned in the table above) tested during this thesis were able to handle the mutated .png files. The images below shows some of the created test cases.

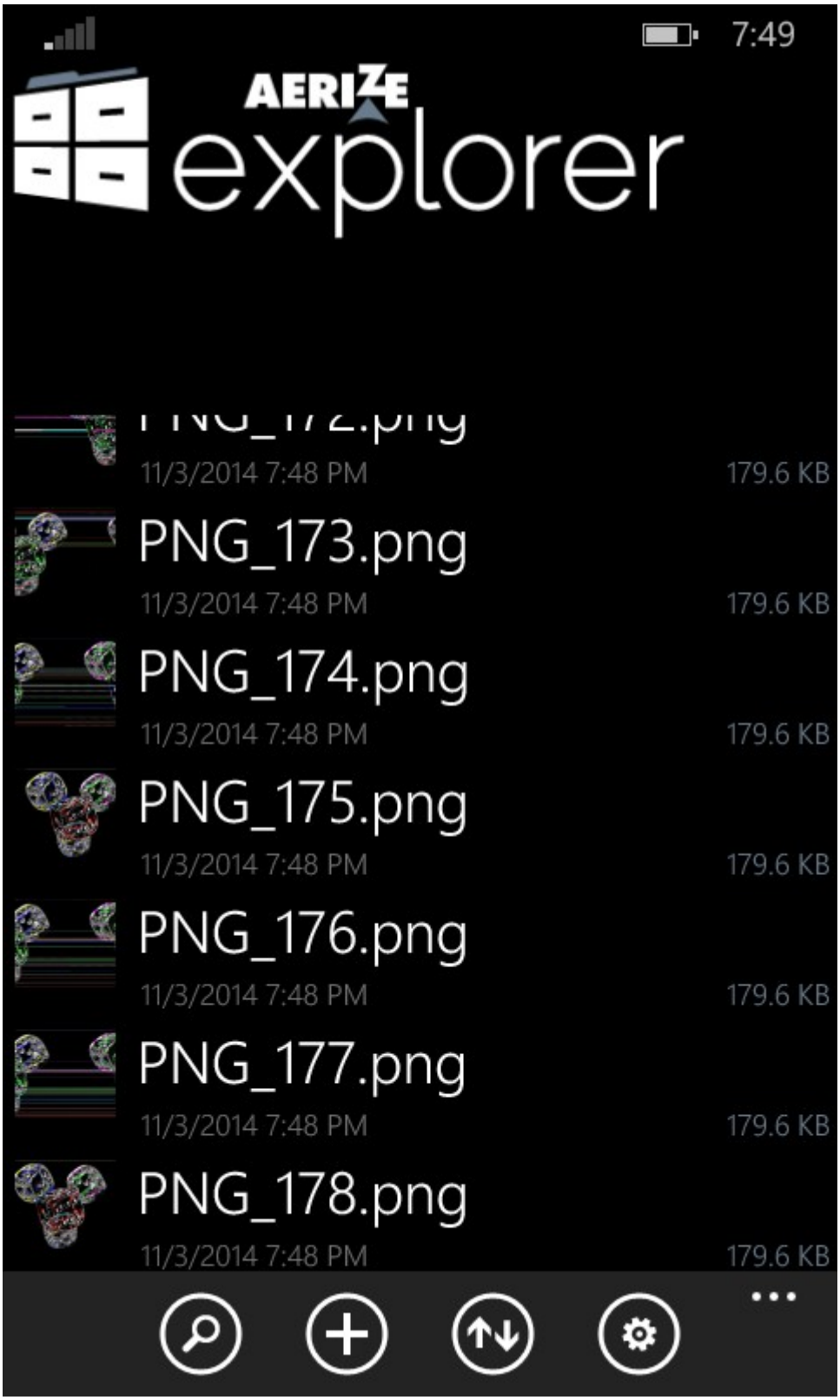


Illustration 23: PNG mutated files

4.2.2 JPEG

Like the PNG files JPEG files were properly executed from the tested applications. The images below show some of the created test cases.

Illustration 24: JPEG mutated files

4.2.3 BMP

The tested applications managed to launch all the .bmp files. As mentioned earlier a crash of the file browsing app occurred (Aerize Explorer) probably because of some kind of overload. After re-launching the app and the test cases that were being launched during the time of the crash nothing happened.

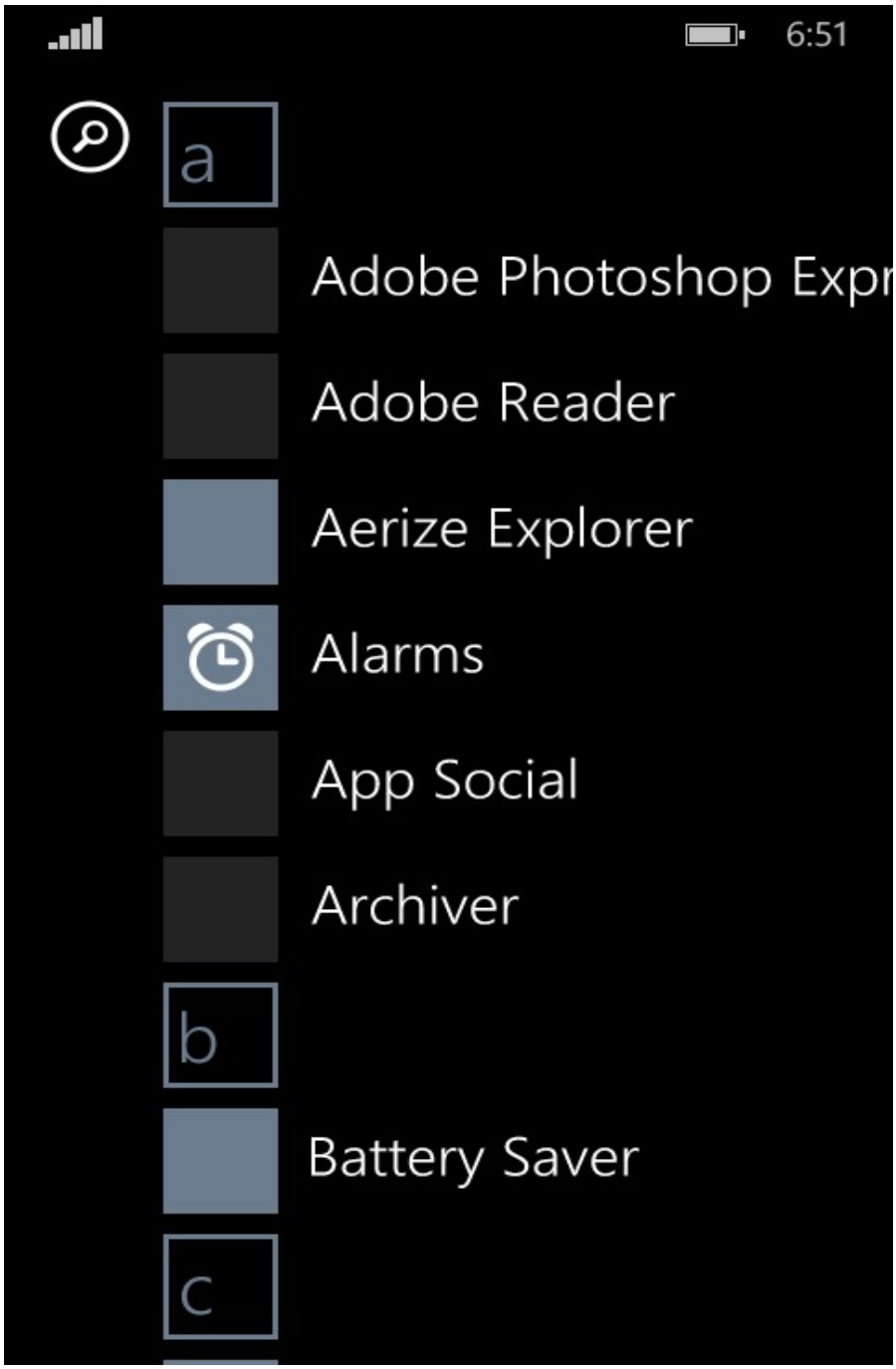


Illustration 25: After the crash

After aerize explorer crashed and exited, delays on the phones functionality were noticed and the apps screen was not showing applications' icons, but without the use of a debugger, further investigation was not possible.

The images below test cases and their creation is showed.

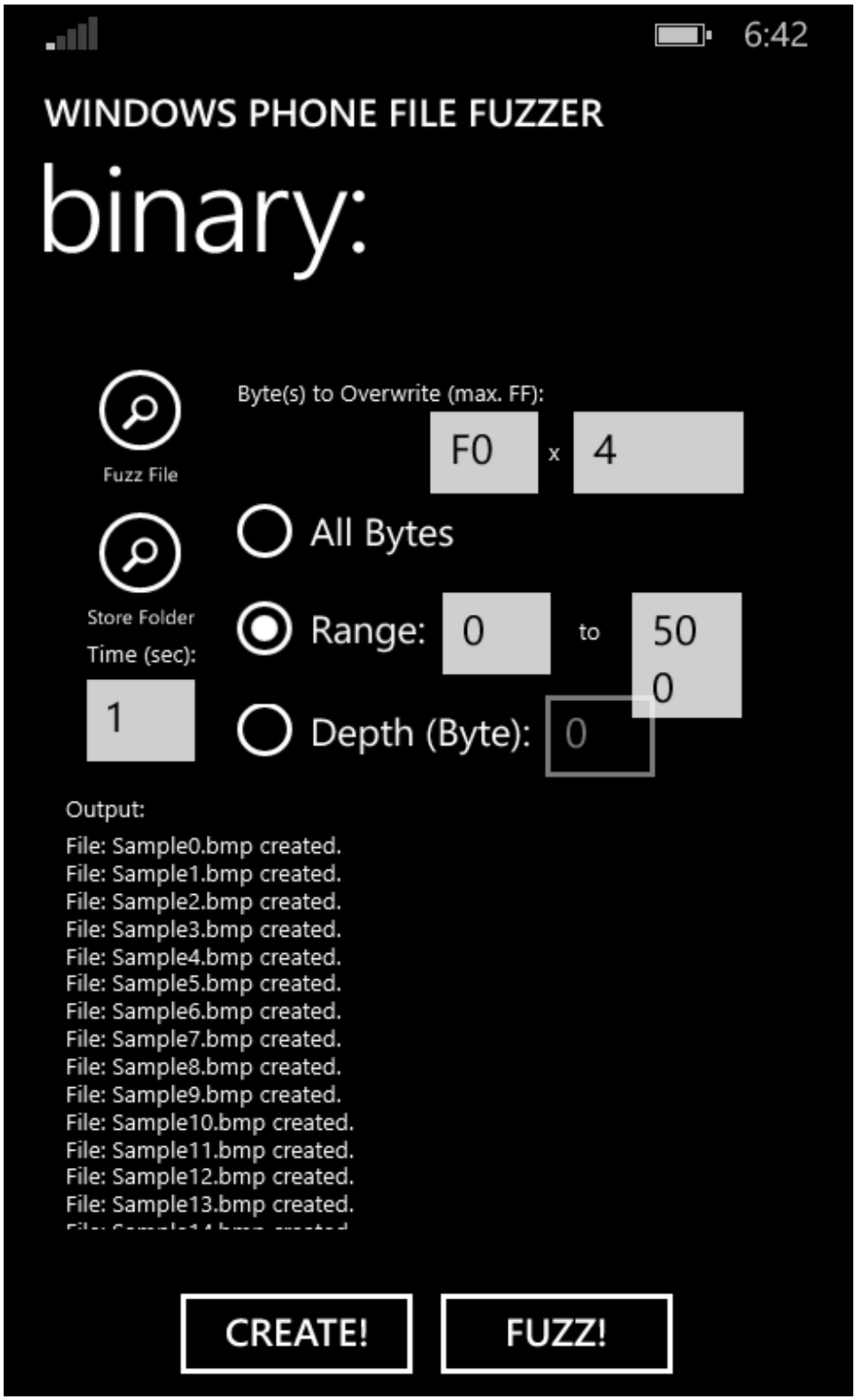


Illustration 26: BMP test cases creation

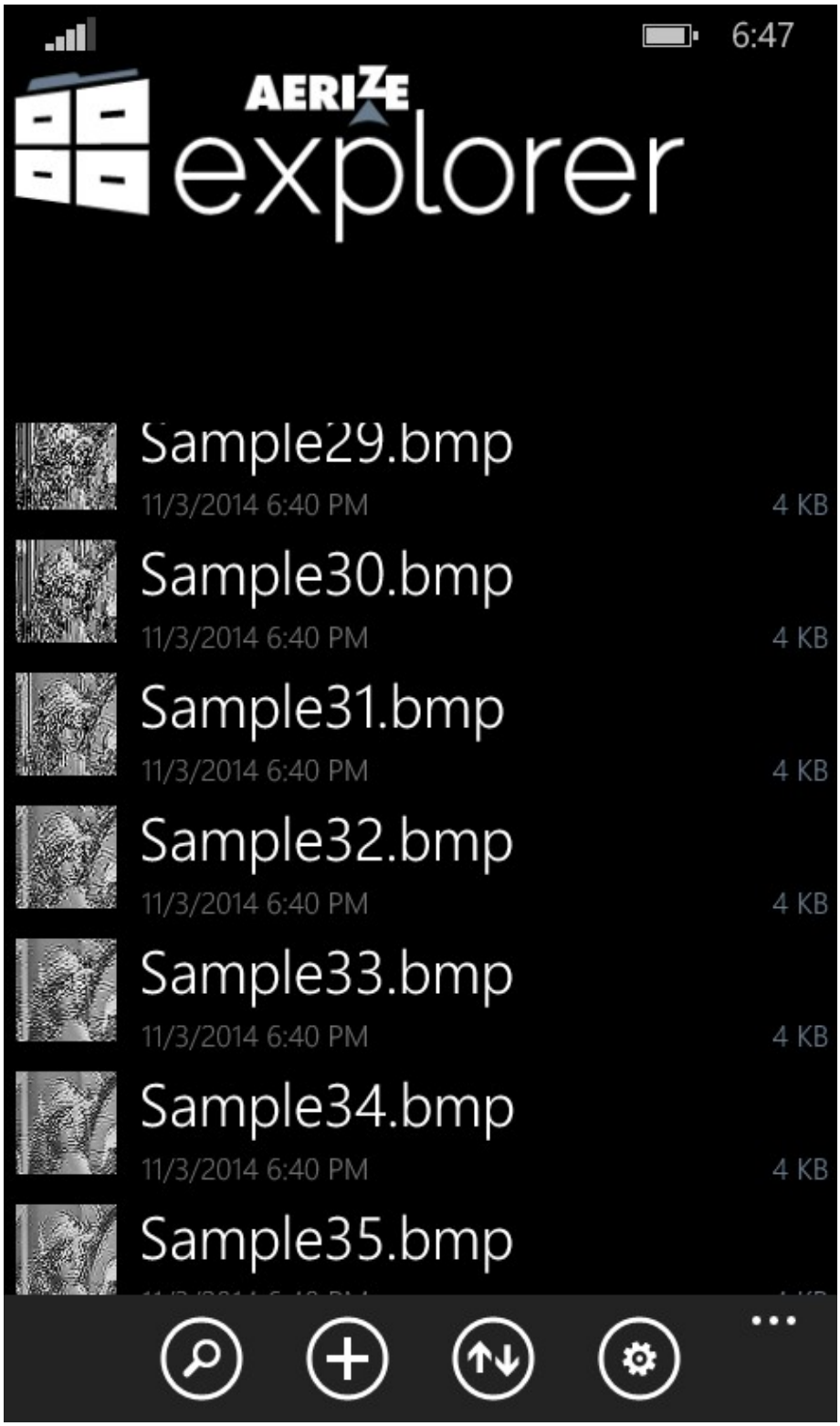


Illustration 27: BMP mutated files

Illustration 28: BMP mutated file opened via Photo viewer

4.2.4 GIF

GIF mutated images were properly launched from the tested applications. Caption were not captured.

4.2.5 DOC

Because of the compressed format of the DOCX files, DOC files were used because it was most possible to create crashes. DOCX files are better fuzzed using generation-based fuzzers. Nevertheless no crashes occurred and Windows Phone Office application properly handled the “corrupted” files.

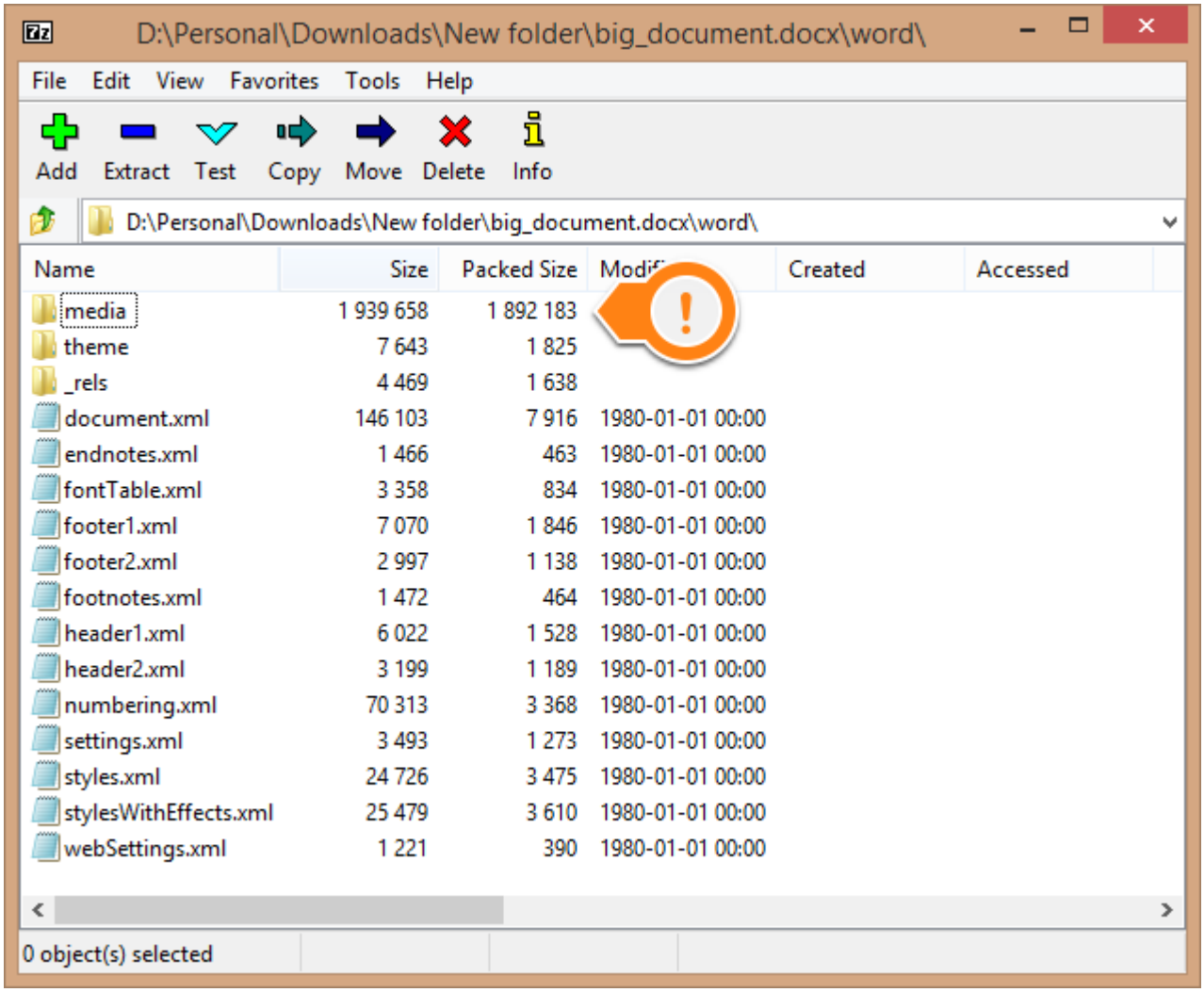


Illustration 29: Uncompressed DOCX file

The pictures below show some of the mutated files and how Office handled them.

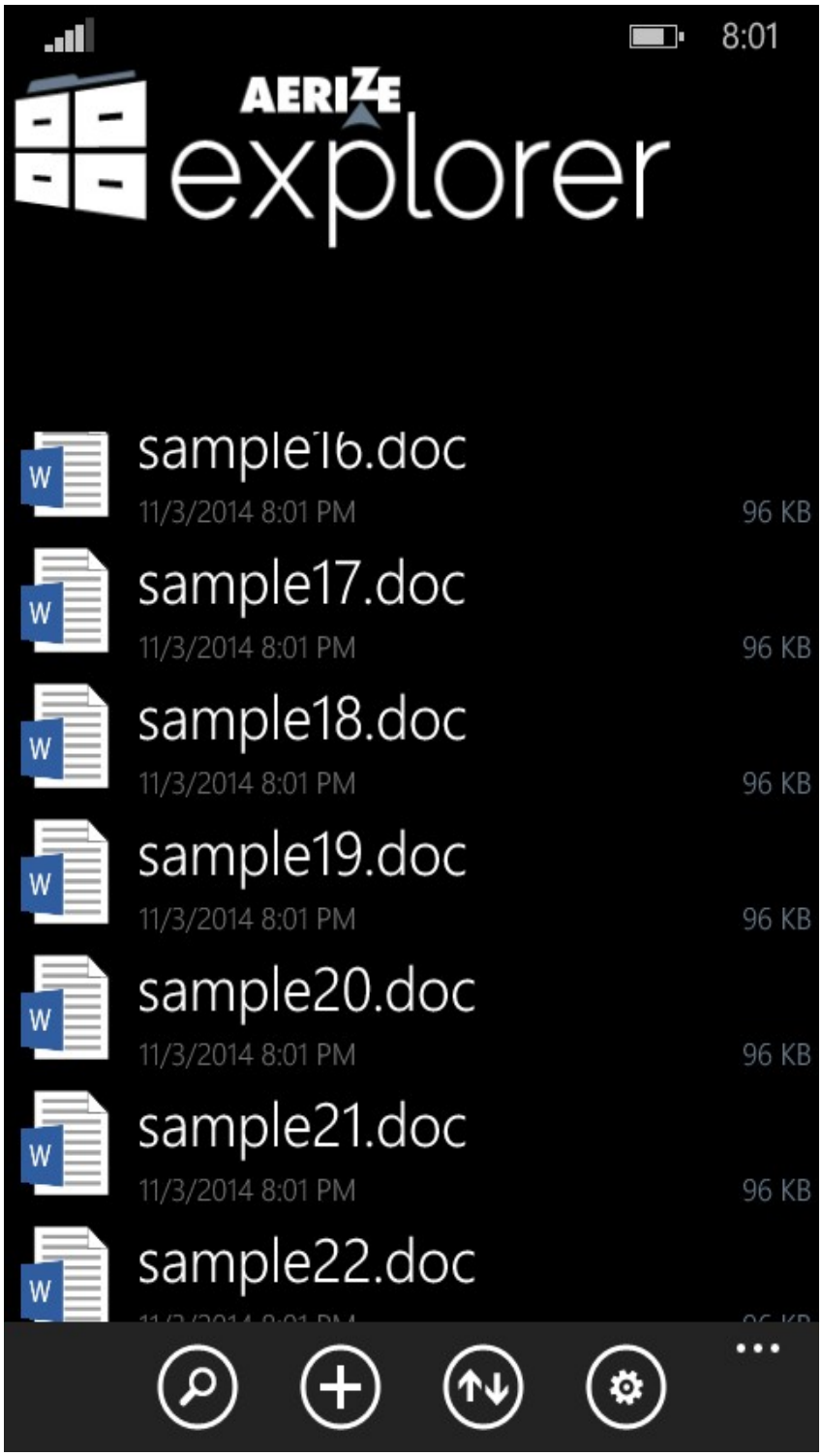


Illustration 30: DOC mutated files

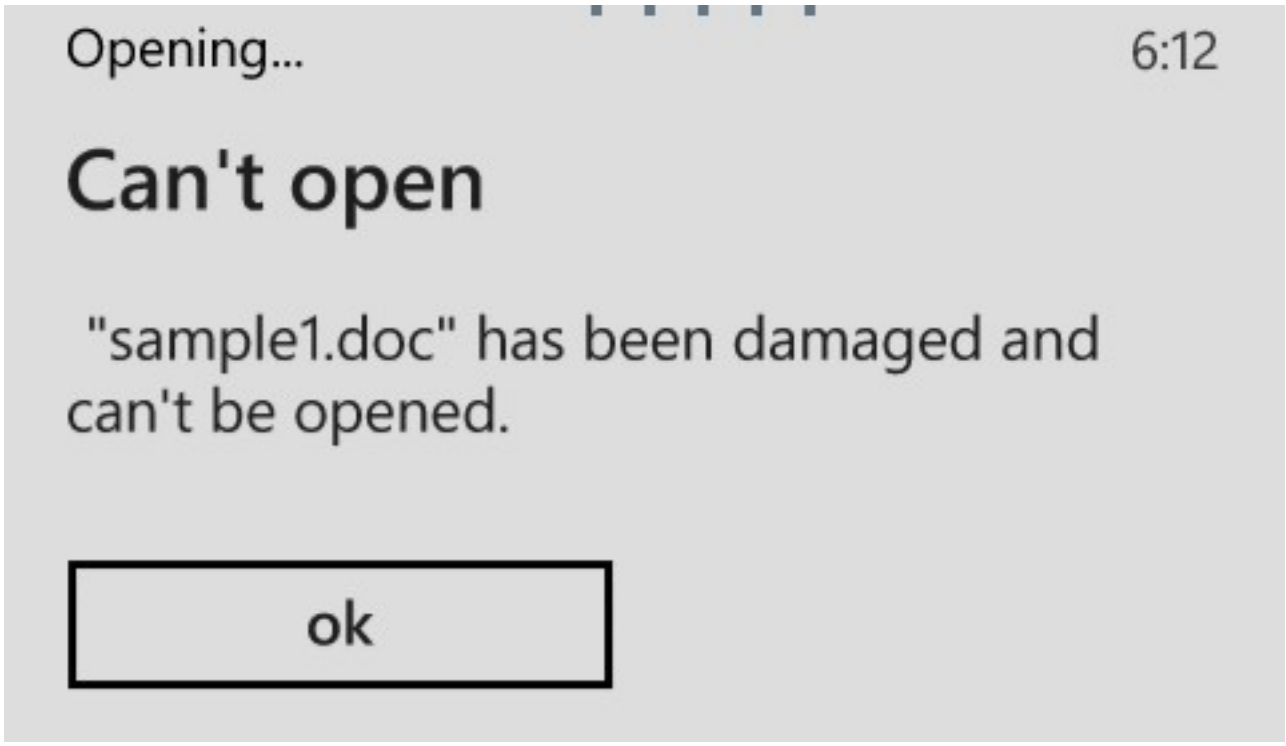


Illustration 31: DOC can't be opened

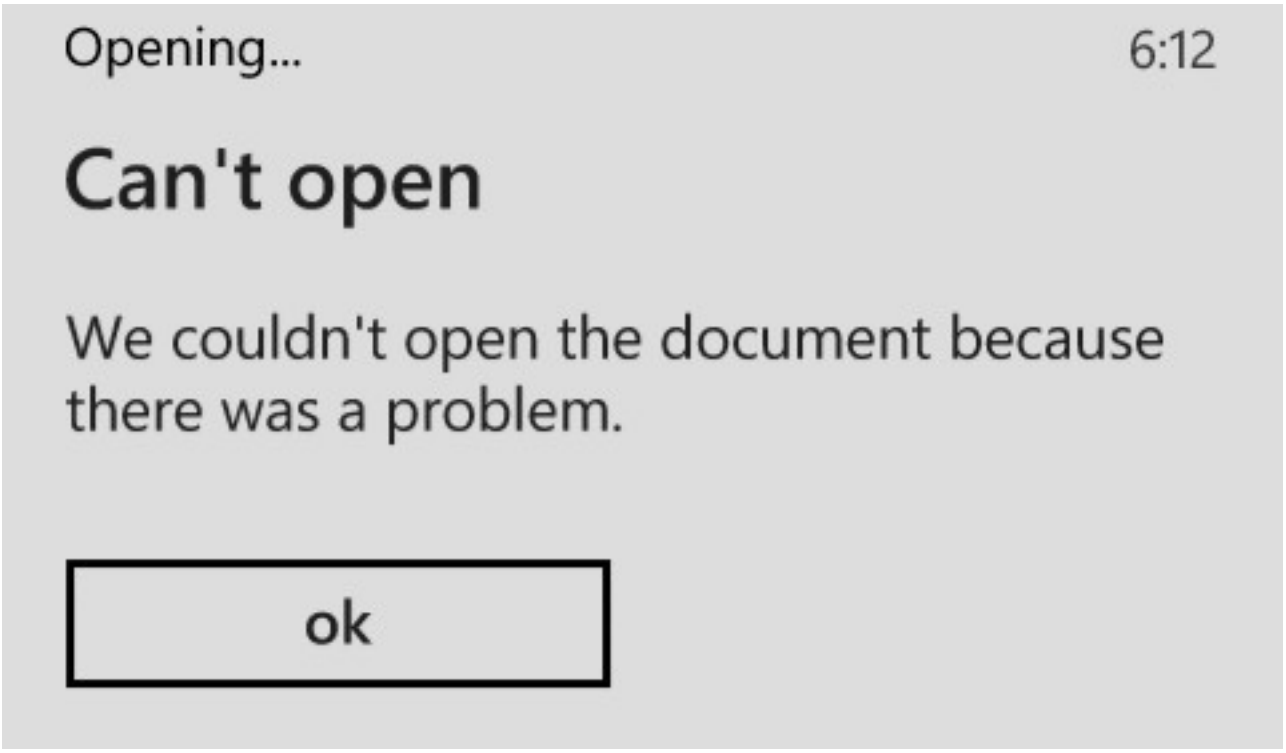


Illustration 32: DOC can't be opened 2

4.2.6 XLS

As explained above for the .doc files .xls was used instead of .xlsx. Office had no problem to handle the mutated files. The picture below shows an error message during an excel file execution.

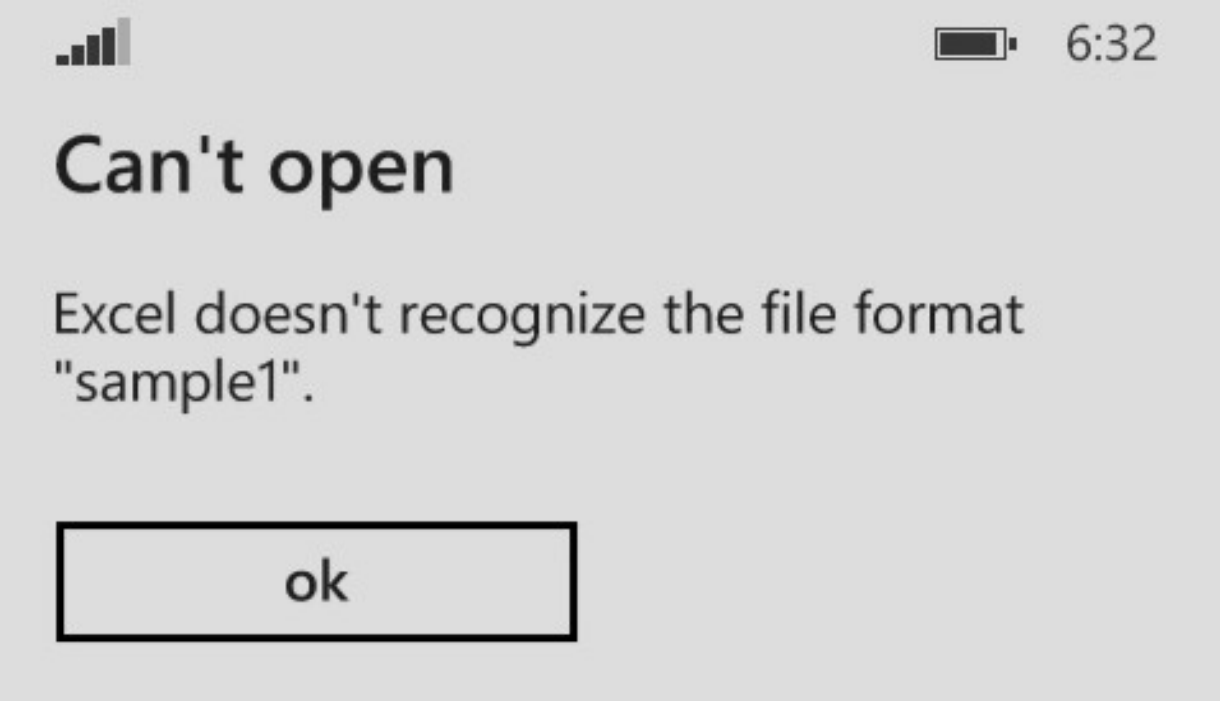


Illustration 33: XLS can't be opened

4.2.7 ZIP

Compressed file types are usually being fuzzed with generation-based fuzzers for better results. However Archiver application was fuzzed using Windows Phone 8.1 File Fuzzer which is a mutation-based fuzzer. No crashes occurred and the application handled the mutated files. The pictures below shows the process of fuzzing the Archiver application.

Most of the files were opened by the Archiver and for some of them error messages were shown.

Illustration 34: ZIP mutated file unable to open

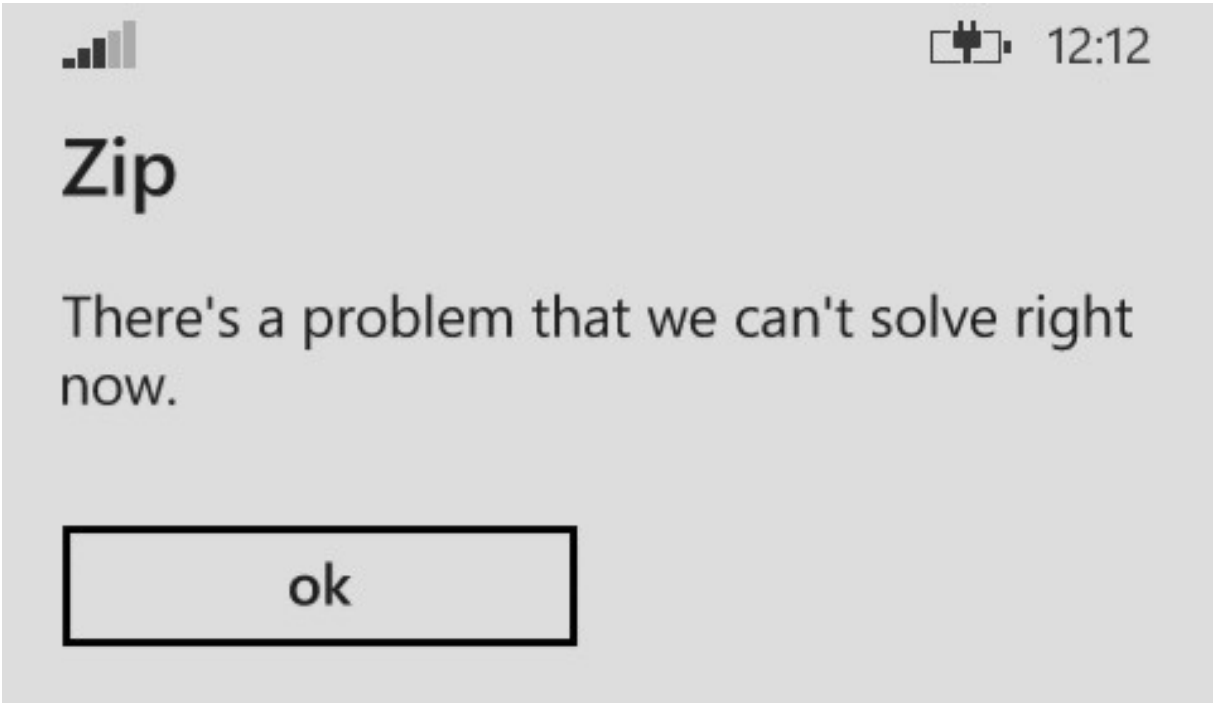


Illustration 35: ZIP mutated file unable to open 2

Illustration 36: mutated ZIP 13

Illustration 37: mutated ZIP 30

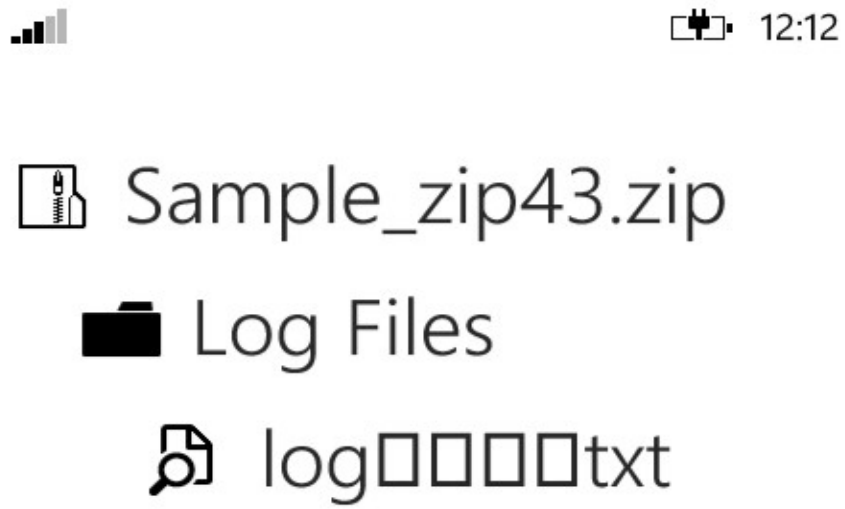


Illustration 38: mutated ZIP 43

4.2.8 PDF

PDFs are also compressed files and it is suggested to be fuzzed using generation-based fuzzers. The mutated files created did not provided any crashes and both Acrobat Reader and PDF Reader managed to handle the test cases. The pictures below show the process of creating and executing the mutated .pdf files.

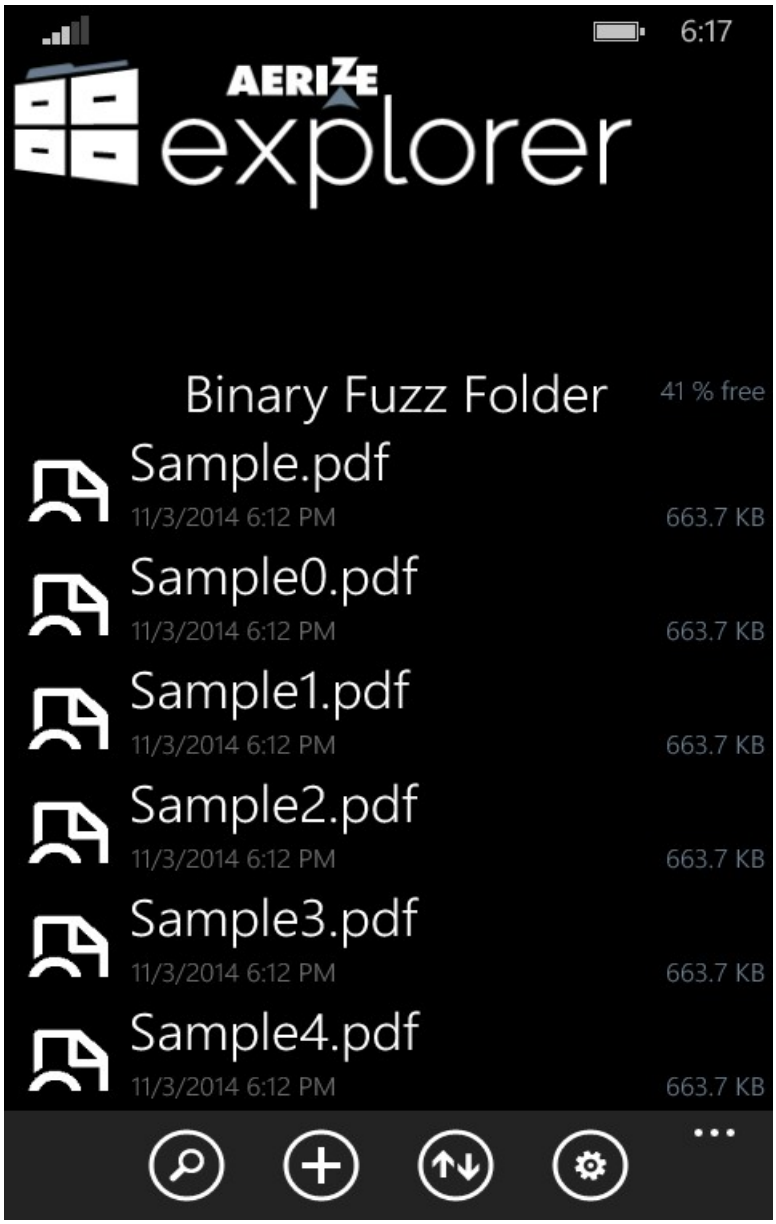


Illustration 39: PDF mutated files

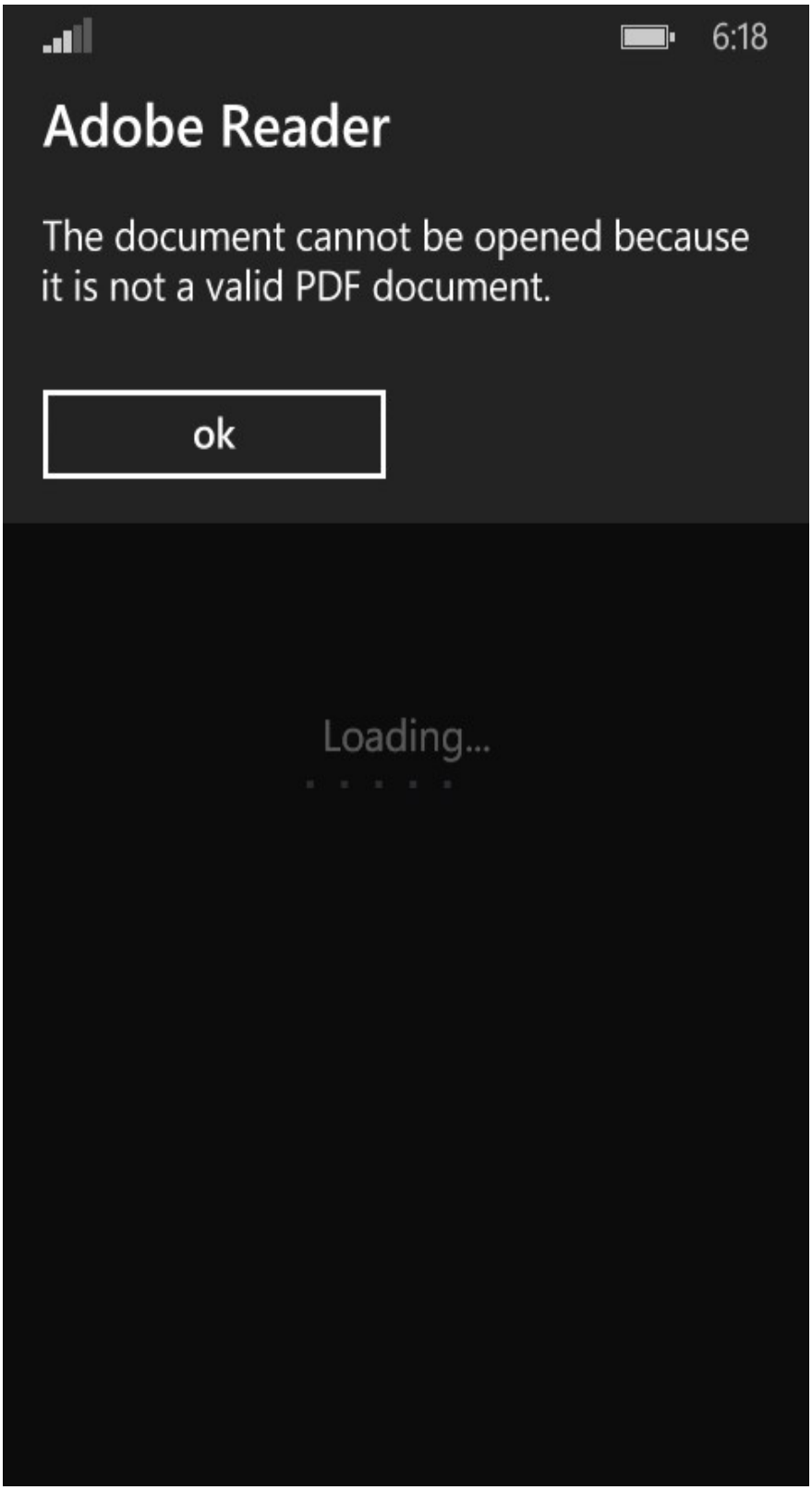


Illustration 40: PDF mutated file not valid

4.2.9 EPUB

EPUB files are also compressed, thus the fuzzing process was not expected to produce crashes. EPUB Reader was able to handle the mutated files. The pictures below shows the mutated files.

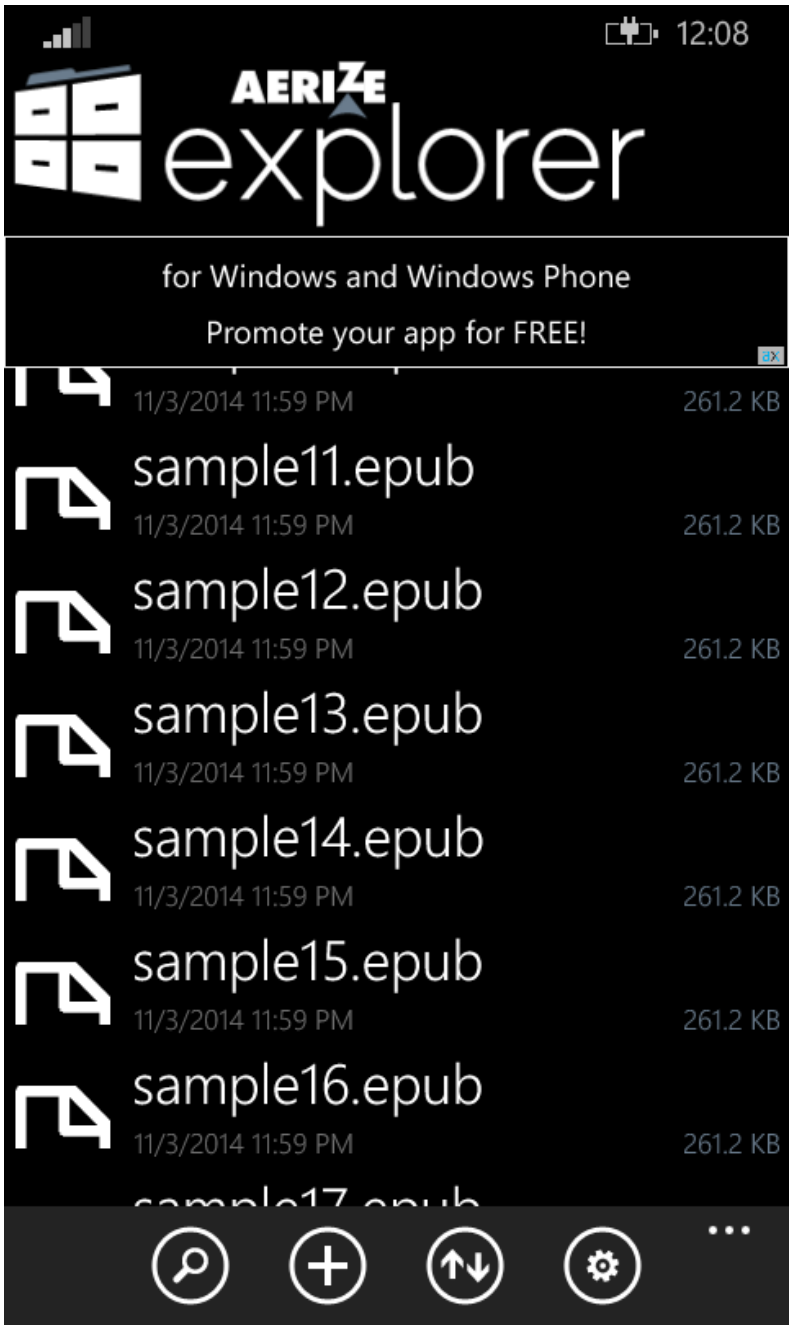


Illustration 41: EPUB mutated files

5. Monitoring

After creating the test cases the most important thing (even more important) is monitoring. Microsoft provides a number of tools for monitoring and debugging. The most interesting are *Windbg* and *Tshell*, provided within Microsoft's OEM tools.

Using Tshell and Windbg a debugger can be attached to a process running inside the testing device. Also Windbg gives the user the opportunity to save logs and dumps in any chosen directory.

Sadly, Microsoft only provides these tools to its Hardware Partners (OEMs), so the monitoring process could not be done in this thesis.

6. Conclusions

This thesis is the first attempt to create a File Fuzzer (as far as known) in a Windows Phone environment and the security measures Microsoft took in it's mobile OS where hard to overcome. The fact that Microsoft don't let 3rd party developers to launch and kill processes makes auto-execution of the test cases nearly impossible. Also the fact that attaching a debugger to a process inside Windows Phone needs tools that are provided only to registered OEM (Hardware constructors) users is a drawback.

The fact that no clear crashes occurred during the fuzzing process indicates that it was not successful, but the lack of automation makes the launching process very difficult and time consuming. Also the fact that a debugger could not be used makes the monitoring process impossible. This does not mean that the tool is useless. Fuzzing is not an exact science and needs a great amount of seed files and test cases in order to get results.

Most definitely the tool would provide a decent amount of crashes if Microsoft had given the opportunity to the developers to handle the file launching operation (this is why the “FUZZ!” button does not work) and of course provided a debugger in order to investigate the crushes.

Nevertheless building a File Fuzzer for this OS was a very educational and interesting experience and it is a starting point for anyone who wants to fuzz or create his own File Fuzzer for Windows Phone.

7. Further Work

While Windows Phone 8.1 File Fuzzer is a functional File Fuzzer, some major improvements need to be done. Some of them are possible others are not, however most of these improvements are important for a fully functional and competitive File Fuzzer.

The major drawbacks where mentioned in the *Conclusions* chapter and fixing them is not something a developer can do, but the improvements mentioned below are some interesting functions the Fuzzer could use.

7.1 Metasploit String Pattern Creator

Metasploit string pattern is “tool” used in metasploit framework which generates a string composed of unique patterns that can be used instead of *A* sequences in order to let the user know in which exact byte the target application crashed.

A metasploit string pattern creator will be a very useful addition to Ascii and Binary Test Case Creation.

7.2 Code Coverage

In computer science, code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.[10]

Code coverage is needed to test how much of the target source code is being tested while fuzzing.

7.3 Generation Based

Generation Based Fuzzing was mentioned in chapter 1. Microsoft gives the developer the ability to create his own custom files. This is an opening for a generation based Fuzzer addition to the existing project.

Using the benefits of Generation Based Fuzzing for known file protocols increases the possibility of finding exploitable crashes.

Most of the leading fuzzing frameworks nowadays has mutation and generation based abilities. Peach fuzzer for example allows testers to create smart fuzzers adapted to their needs through XML configuration files called “Peach pit files”. Making such files needs knowledge of the format message and state machine of the targeted protocol as well as the actor Peach has to fuzz.

```
<!-- Defines the format of a WAV file -->
<DataModel name="Wav">
    <!-- wave header -->
    <String value="RIFF" token="true" />
    <Number size="32" />
    <String value="WAVE" token="true"/>
</DataModel>
```

Illustration 42: Data Model XML for Peach Fuzzer

7.4 Debugger

Finally, a debugger must be provided by Microsoft to help the developers and/or the testers to determine if a crash is vulnerable or not.

The problem with the current SDK is that Microsoft does not provide the Classes/Methods to develop a debugger.

Bibliography

[1]. Michael Sutton, Adam Greene, Pedram Amini. "*Fuzzing: Brute Force Vulnerability Discovery".* Addison-Wesley, July 9, 2007.

[2]. Codenomicon. *The Buzz on Fuzzing.* http://www.codenomicon.com/products/buzz-on-fuzzing.shtml

[3]. Wikipedia. *Fuzz Testing*. http://en.wikipedia.org/wiki/Fuzz_testing

[4]. INFOSEC Institute. *Fuzzing-Mutation vs. Generation.* http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/

[5]. OWASP. *Fuzzing.* https://www.owasp.org/index.php/Fuzzing

[6]. Wikipedia. *Windows Phone. http://en.wikipedia.org/wiki/Windows_Phone*

[7]. Microsoft. *Windows Phone Architecture Preview.* https://dev.windowsphone.com/en-us/OEM/docs/Getting_Started/Windows_Phone_architecture_overview

[8]. Wikipedia. *Windows Runtime.* http://en.wikipedia.org/wiki/Windows_Runtime

[9]. Microsoft. *Windows Phone 8.1 Security Overview.* Microsoft, April 2014.

[10]. Wikipedia. *Code Coverage. http://en.wikipedia.org/wiki/Code_coverage*

[11]. Collin Mulliner, Charlie Miller. *Fuzzing the Phone in your Phone.* Black Hat USA, June 25, 2009.

[12]. Charles Miller. *How smart is Intelligent Fuzzing-or-How stupid is Dumb Fuzzing.* Independent Security Evaluators, August 3, 2007.

[13]. Charlie Miller, Zachary N. J. Peterson. *Analysis of Mutation and Generation-based Fuzzing.* Independent Security Evaluators, March 1, 2007.

[14]. Kevin Mahaffey, John Hering, Anthony Lineberry. *Fuzzit: A Mobile Fuzzing Tool.* DEF CON 17, 2009.

[15]. Wouter Veugelen. *Windows Phone 8.* Mobile Application Penetration Testing, 2013.

[16]. Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, David Brumley. *Optimizing Seed Selection for Fuzzing*. 23Rd Usenix Security Symposium, August 20-22, 2014.