SS008

# Generative Grammar-Based Fuzzing

# Research Plan

## 1. Rationale

Despite technological and code advancements, software vulnerabilities remain a threat to the cyber world. Nowadays, web browsers have become the primary attack vector for attackers. In recent years, there have also been numerous cyber attacks, including the theft of Singaporeans' medical data in the singhealth attack, and the use of the eternal blue vulnerability in the WannaCry ransomware attack. Furthermore, software vulnerabilities are extremely costly to fix if exploited, with estimated losses of $4 billion from the 2017 WannaCry ransomware attacks. Thus, vulnerability discovery is our utmost priority.

This project aims to develop a workflow to find bugs in browses via fuzzing. To do so, we will be using the open source project, Domato. Domato is a DOM fuzzer written by Ivan Fratric in 2017 to fuzz web browsers. It has found 31 CVEs across popular web browsers Apple Safari, Google Chrome, Microsoft Internet Explorer, Microsoft Edge, and Mozilla Firefox. Domato is a generative grammar-based fuzzer that generates random but synthetically valid HyperText Markup Language (HTML) files. These files are to be opened by the web browser in hopes that buggy sections of code within the browser will lead to crashes upon running some of the HTML code. To test the validity of our workflow, we decided to fuzz an older version of Mozilla Firefox (53.0).

## 2. Research Question(s)

- How to find bugs in web browsers?
  - What techniques can we use to find bugs in web browsers?
    - What fuzzer can we implement?
    - How can we modify or extend the fuzzer?
  - Which web browser should we use to test the programme?
  - How does one catch crashes in a programme?

## 3. Hypothesis

We hypothesize that our modified fuzzer will be able to find new bugs in web browsers.

## 4. Engineering goal(s)

Our aim is to modify and improve the grammar in the Dom fuzzer, Domato, developed by Ivan Fradric. We hope to improve on the Cascading Style Sheets (CSS), Scalable Vector Graphics (SVG) sections of Domato. Furthermore, we hope to use the existing grammar engine within Domato to create an eXtensible Stylesheet Language Transformation (XSLT) fuzzer.

In addition, we hope to create a fuzzing programme to run Domato and concurrently, check for and record bugs on a single machine.

## 5. Expected Outcome(s)

We hope to be able to use the software, Domato, to find vulnerabilities in the web browsers. In our case, we will be using Firefox to test our fuzzer.
For each bug we find, we hope to be able to extract the specific piece of code which causes the bug (to help others reproduce the bug), understand how the bug causes the crash.

## 6. Procedures

We analyse (statically and dynamically) the structure of Domato to understand how it creates synthetically valid HTML files using the grammar provided. We identify areas of the grammar or code that could be improved. A web crawler is used to download example files (HTML, CSS or SVG files) from the internet. A python script utilising regular expressions is written to extract the necessary grammar from the files and the new grammar is added into Domato. We also write new grammar to generate random but synthetically valid XSLT files to do XSLT fuzzing.

During/ after this improvement, we write a command script to call Domato to generate the files and then call firefox to open the files while being monitored by the open source crash detection programme, BugId. In the event we encounter a crash, we use BugId to generate a report of the crash and we analyse the HTML file understand the nature of the crash.

After finding a few buggy files, we use a script and eyeballing to minimise the file to remove redundant lines of code to enable us to perform root cause analysis.

## 7. Risk and Safety

- Don’t connect the computer to the Internet when fuzzing to ensure bugs are not directly reported to the programme manufacture

- Ensure that the computer does not overheat during the fuzzing (can be done by monitoring the temperature of the CPU)

## 8. Methods for Data Analysis

Improving Domato phase

- Using a web crawler (HTTrack) to find necessary files
- Using regular expressions to search for necessary portions of the downloaded files
- Using regular expressions to substitute specific values for general values

Fuzzing Phase

- Run BugId to identify the bugs for further analysis.

Analysis Phase

- Batch and python scripts to triage bugs
- Python script to minimise buggy files for root cause analysis
- Static analysis to identify the root cause of a bug

## Bibliography from your literature review

1. Ivan Fratric. Domato. (last updated, 2018, July 04). Googleprojectzero/domato. Retrieved from https://github.com/googleprojectzero/domato
2. SkyLined. BugId (accessed 2018, November 21). SkyLined/BugId. Retrieved from https://github.com/SkyLined/BugId
3. HTML. (n.d.). Retrieved from https://www.w3schools.com/

## Abstract

The prevalent use of web browsers makes it a high-value target for hackers attempting to gain access to a target's system. A key to browser security is for vulnerabilities in browsers to be promptly discovered and patched. A common method to uncover vulnerabilities is by fuzzing, where random but synthetically valid inputs are generated, and executed by the targeted software so that vulnerabilities are revealed as software crashes when buggy code processes such unexpected input. In this project, we developed a workflow for fuzzing browsers. Using this workflow, we extended Domato[4], a grammar-based fuzzer, by hunting for rarely used CSS and SVG components and incorporating it. Using Domato, we also wrote an XSLT fuzzer from scratch. Using BugId[5] for crash detection, we tested our fuzzer on Firefox (version 53.0) for 11 days, executing 35500 generated files and caused 296 crashes. Triaging was then performed, and 3 unique crashes were identified – two stack exhaustion and one denial-of-service.

# Report

## 1 Introduction

In the face of theoretical and practical advances in code development, software vulnerabilities is nonetheless still an ineradicable security problem in all non-trivial programmes. Software vulnerabilities have, in recent years, have led to financial losses and the loss of time.

Examples of recent cyber attacks would include the theft of Singaporeans' medical data in the SingHealth attack and the use of the eternal blue vulnerability in the WannaCry ransomware attack in 2017 that resulted in an estimated loss of $4 billion.[1] As such, an increasing number of people are engaging in bug hunting and vulnerability analysis to uncover new potentially exploitable vulnerabilities.

Bugs in browsers are especially significant because they affect a large number of people. This is especially so for major browsers such as Firefox which has a market share of 10.96% of all browser usage in August 2018[2]. If left unpatched, these bugs could be used by malicious parties to gain access to organizations' or individuals' systems and changes could be made to the system regardless of the geographical location via remote code execution. After gaining access to the system, the attacker could run commands to leave a persistent backdoor/shell on the victim to repeatedly control their computer. They could also force the victim's computer to upload sensitive classified information. Finding these vulnerabilities and reporting them to the programme's manufacturer allows the manufacturer to roll out patches and prevent these vulnerabilities from being exploited. Hence, many manufacturers pay good money for them. Google rewards as high as $15000 for a sandbox escape vulnerability in chrome.[3]

One such method of bug hunting would be 'fuzzing'. Fuzzing, also known as fuzz testing, is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program with the hope that these inputs will expose vulnerabilities within the computer program. These vulnerabilities could be exposed in the form of a crash or a hang.

In this paper, we attempt to build a workflow to improve a fuzzer and perform fuzzing on a web browser to find new bugs. This would consist of understanding, improving and setting up a fuzzer and debugging tools to identify crashes and hangs, followed by analysis of the results of the fuzzer. The fuzzer that we have decided to improve on is Domato[4]. Domato has found 31 CVEs across popular web browsers Apple Safari, Google Chrome, Microsoft Internet Explorer, Microsoft Edge and Mozilla Firefox. The debugging tool we chose was BugId[5] as it is not only able to detect crashes in browsers but is also able to generate reports about each crash, this is useful for triaging and bug analysis. To test the validity of our workflow, we decided to fuzz an older version of Firefox (53.0).

### 1. 1 Related Work

Fuzzing has been used since the 1980s and is a staple in modern vulnerability testing. There are several different types of fuzzer for different fuzzing approaches.[6]

Fuzzers can also be grammar-based or non-grammar based. Grammar-based fuzzers generate an input based on a known syntax whereas non-grammar based fuzzers generate input without a format. While non-grammar based fuzzers may seem to generate more varied input, it is often useful to generate input based on a specified syntax to ensure that an application's parser accepts the input as valid to pass to the programme.

Fuzzers can be generative or mutative. Generative fuzzers creates inputs from scratch while mutation based fuzzers generate new input by modifying existing inputs. Mutative fuzzers are generally simpler to set up because it only requires a few specific input examples to start running whereas one might require spending time writing the input structure for a generative fuzzer, particularly if it is also grammar-based, however, they are limited by the input examples. An example of a mutative fuzzer would be the popular radamsa fuzzer.[7]

Some fuzzers are white-boxed fuzzers which aware of a programme's structure (such as American Fuzzy Lop(AFL)) and thus try to generate input to traverse the different control flow branches. Others such as Zuff[8] are black-box fuzzers and generate input without consideration of programme structure. While whitebox fuzzers tend to give better results, they are also more difficult to set-up. Instead of having the fuzzer monitor the target application to gain greater code, one can also analyse the source code (if the target is open source) or perform reverse engineering to better understand a programme's structure and write a specialised fuzzer specifically to generate interesting input for that application.

Domato itself is a generative, grammar-based, black-box fuzzer.

## 2 Materials and Methods

Equipment needed:

1. A 32/64-bit computer running Windows 10, preferably with a fast CPU and SSD

Prerequisite Software:

1. Python 2.7.16 (to run BugId and Domato)
2. Python 3.4+ (to run our scripts)
3. Domato[2]
4. BugId[3]
5. HTTrack Webcrawler[9]
6. Firefox (53.0 and above)

### 2.1 Improvements Made to Domato

Before setting up the fuzzing system to run Domato, we made several improvements to Domato in order to increase our chances of finding new bugs. There are several components to the input code generated by Domato:

1. HyperText Markup Language (HTML)
2. Cascading Style Sheets (CSS)
3. JavaScript (JS)
4. Scalable Vector Graphics (SVG)

In this project, we decided to focus on extending the CSS and SVG grammar components of Domato. Furthermore, we also extended Domato by writing a grammar for eXtensible Stylesheet Language Transformation (XSLT) code, which is not native to Domato. XSLT has a history of finding exploitable bugs in Firefox (e.g. CVE-2017-5376 and CVE-2010-1199), thus we decided to implement XSLT.

### 2.1.1 Cascading Style Sheets(CSS) & Scalar Vector Graphics(SVG)

CSS is a programming language used to describe to a browser how a certain webpage should be styled. SVG is an XML-based vector image format for two-dimensional graphics and is used to create images.

To improve Domato's CSS and SVG code generation, we attempted to find new CSS properties and their corresponding values as well as new SVG elements, attributes and their corresponding values to incorporate into Domato. Doing so allows Domato to generate more varied CSS and SVG code, increasing our chance of finding new vulnerabilities.

With reference to *fig 2.2,* we started by using HTTrack[9] to crawl the web for CSS, SVG and HTML files (web developers sometimes use inline SVG in their HTML code). We started the crawler from multiple different sites to ensure that there were a variety of CSS, SVG and HTML files downloaded.

Before extracting the SVG elements, attributes and values, we had to extract the SVG section of code from the HTML files as directly extracting the SVG components from the HTML files using regular expressions would end up extracting HTML components as well.

To extract the CSS properties from the CSS files, we used a python script with regular expressions. To extract the CSS properties, the regular expression searched for words between '**{**' or '**;**' and '**:**'. After this, we used a python script to compare the CSS properties we extracted with the existing CSS properties in Domato to get the missing properties. To extract the SVG elements and attributes, this process is repeated with different regular expressions.

To extract the CSS values of the missing CSS properties, the regular expression searched for words between **'property:**' and '**;**' or '**}**'. This process is also repeated to extract the SVG values for the missing SVG attributes.

When extracting the CSS and SVG values, we noticed that some of the values were specific examples of general values. E.g. `#AABB6A` is a code for a colour. Thus, we had to use more regular expressions to replace said specific values with their general symbols. For example, `#AABB6A` would be converted into `<color>`.

Subsequently, the missing CSS properties and their respective values are formatted properly and put into text files to be read by the Domato grammar engine. These text files are included into Domato's generator. This process is repeated with the SVG elements, attributes and values. The new CSS properties, SVG elements and attributes were weighted to appear as 50% of all the CSS properties, SVG elements and SVG attributes generated. This helps increase our chances of finding bugs with our modified fuzzer.

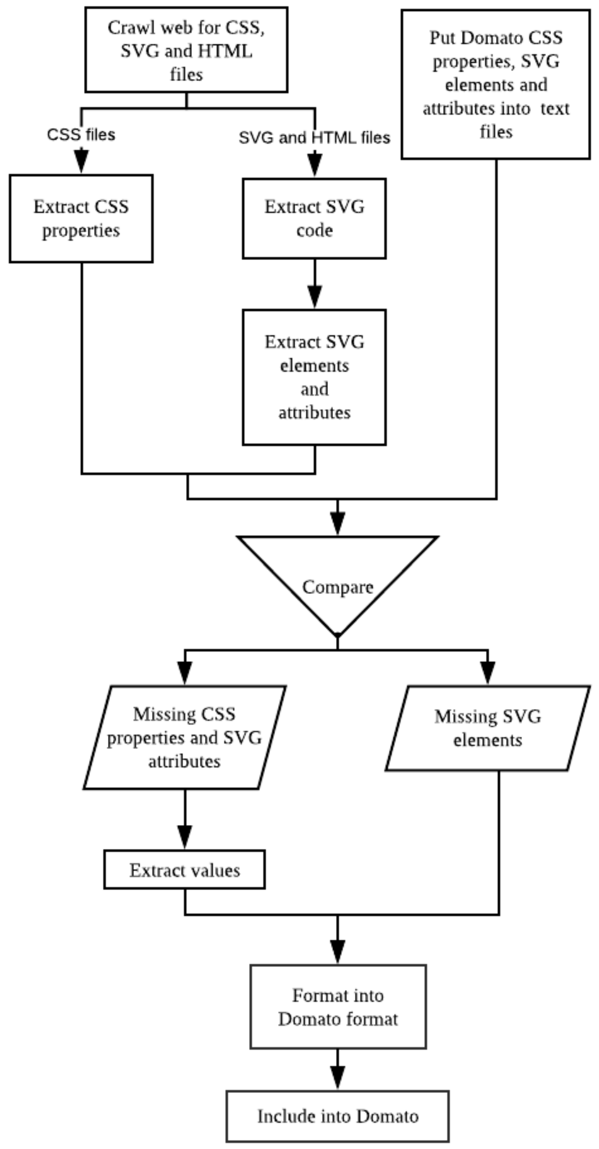
Crawl web for CSS, SVG and HTML files
CSS files
SVG and HTML files
Put Domato CSS properties, SVG elements and attributes into text files
Extract CSS properties
Extract SVG code
Extract SVG elements and attributes
Compare
Missing CSS properties and SVG attributes
Missing SVG elements
Extract values
Format into Domato format
Include into Domato

*Fig 2.0: Workflow for extending*

### 2.1.2 eXtensible Stylesheet Language Transformation (XSLT)

XSLT is a language used to transform eXtensible Markup Language (XML) documents into other XML documents or formats such as XHTML to be displayed on web pages. The XSLT grammar we wrote transforms a prewritten, static XML document into an XHTML document to be displayed by Firefox.

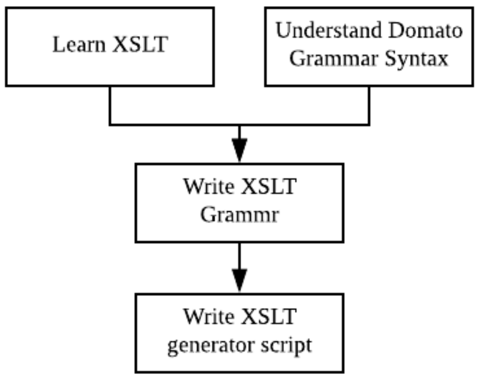

*Figure 2.1: Workflow for extending Domato with XSLT*

To create an XSLT fuzzer, we needed to learn how to code in XSLT. This was done by reading XSLT documentation on websites such as tutorialspoint[10] and W3Schools[10]. Following which, we had to understand Domato's syntax for its context-free grammar via reading its documentation and experimenting with generating symbols.

Following that, we proceeded to write the grammar for XSLT. Some of the elements were difficult to introduce as we had to store the name of previously coded variables and templates before calling them. To achieve this, we had to include python code into the grammar file. Following which, we attempted to link the written HTML, CSS and JS elements into the XSLT file. However, as the parser for XHTML is stricter than that for HTML, including the HTML and JS code generated by Domato lead to parsing errors. The only component that could be included was the CSS component.

Lastly, as we were generating XSLT files using our own grammar file, we could not use Domato's built-in HTML file generator and had to create our own python script to call Domato to parse the grammar file and generate multiple XSLT files.

## 2.2 Flow of the Fuzzing Architecture

We wrote a batch script to automate the generation and testing of HTML and XSLT files. The HTML files consist of HTML, JS, CSS and SVG code whereas the XSLT files only contain XSLT and CSS code.

As *figure 2.3* suggests, the programme starts by generating <u>100</u> HTML files. Generating many files at once is more efficient than generating one file at a time as Domato only needs to parse the grammar once.

Before the testing starts, we need to turn on full page heap for Firefox. When it is enabled, each heap Firefox allocates is placed on the end of a memory page boundary, and the subsequent page is marked as PAGE_NOACCESS. Any buffer overruns are thus registered as access violations and are immediately caught by BugId.

For each HTML file to test, the programme uses Firefox to open the HTML file which is next in line for testing. While this happens, it is running BugId and monitoring for crashes on Firefox. After 30 seconds, the page should have loaded, thus the programme closes BugId and Firefox.

If a bug is indeed found, the programme will copy the HTML file to another directory for future analysis.

A similar flow is used for fuzzing of the generated XSLT files. The main difference is that for each XSLT file, a new XML file linking to that XSLT file will be created.

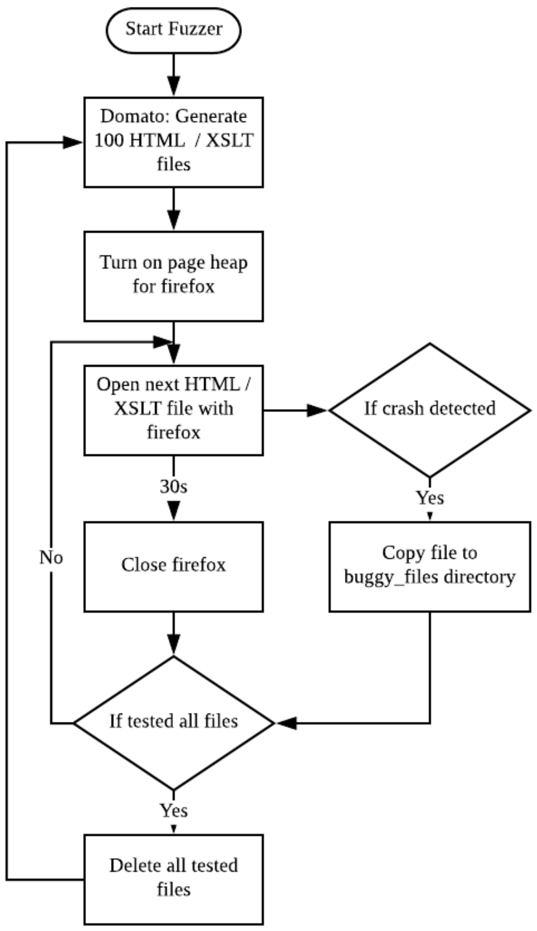
Start Fuzzer
Domato: Generate 100 HTML / XSLT files
Turn on page heap for firefox
Open next HTML / XSLT file with firefox
If crash detected
30s
Yes
Close firefox
Copy file to buggy_files directory
No
If tested all files
Yes
Delete all tested files

*Fig 2.3 Flow chart of fuzzing program*

## 2.3 Triaging Bugs

From the multiple crashes which we obtained, many of them were caused by the same bug (refer to *table 3.0*). Thus, triaging was needed to remove duplicate crashes which were caused by the same bug to allow us to more efficiently look for unique and interesting bugs.

Triaging is a 2-step process:

1. Generating a BugId report of each crash
2. Sorting reports by ID to group crashes utilising the same vulnerability

For each XSLT and HTML file which caused a crash, we run BugId on them again. This time, we set

BugId to generate an HTML bug report for each of the files. This is done using a batch script.

BugId generates an ID for each crash based on the type of bug causing the crash. When the reports of identical bugs are generated, only one of them will be saved, eliminating exact duplicate bugs.

However, this only works for identical bugs where the stack is the same when the crash happens. There are cases where 2 files utilise the same vulnerability to cause a crash but are slightly different in implementation, thus they will be treated as different bugs by BugId. Therefore, we also need to sort such reports and group crashes that utilise the same vulnerability together. This was done using a python script.

The python script would extract the bug type from each of the reports and copy the report and corresponding buggy file into a directory with the name of the bug type.

## 3 Results

*Table 3.0: Overall Fuzzing Statistics*

| File format | HTML | XSLT |
|---|---|---|
| Time fuzzed | 11 days | 11 days |
| No. of files generated | 15000 | 20500 |
| No. of crashes | 3 | 293 |
| No. of unique crashes | 1 | 2 |

*Table 3.1: Crash 1 (XSLT)*

| | |
|---|---|
| BugId | RecursiveCall 73c |
| Location | firefox.exe!xul.dll+0x1872270 |
| Security impact | Denial of Service |
| Description | A recursive function call exhausted available stack memory |

*Table 3.2: Crash 2 (XSLT)*

| | |
|---|---|
| BugId | Breakpoint 1b1 |
| Location | firefox.exe!xul.dll+0x8488BF |
| Security impact | Denial of Service |
| Description | A breakpoint has been reached |

*Table 3.3: Crash 3 (HTML)*

| | |
|---|---|
| BugId | RecursiveCall 73f |
| Location | firefox.exe!xul.dll+0x501A8D |
| Security impact | Denial of Service |
| Description | A recursive function call exhausted available stack memory |

## 4. Discussion

Although crashes are usually indicative of bugs in programmes. This is not always true,  some crashes are merely false positives. Crashes 1 and 3 are examples of such false positives. The problem of stack exhaustion is generally considered to be caused by a bug in programmer code (in this case the XSLT and HTML code Firefox executed) and not a bug in the programming code of the application (Firefox). This is because stack exhaustion is caused by firefox doing what is intended of it, executing functions as it has been instructed to by the XSLT or JS code. The problem of stack exhaustion is a stability issue and not a security vulnerability. Such issues cannot lead to remote code execution and privilege escalation cannot be achieved, thus such issues have a low exploitability.

Crash 2 is caused by a breakpoint in the Firefox code being reached. Breakpoints in the code are lines that intentionally crash a programme when run. Despite these being used during development. Such breakpoints should be removed from the code before release, thus this would be considered as a bug. Furthermore, a breakpoint being reached may imply that the issue is a known problem. This means that this crash is caused by a legit bug in Firefox. However, it is not exploitable and thus not a security vulnerability.

## 5. Limitations and Future Work

While doing this project, we had to deal with several constraints. The main two of which was computing resources and time. We did not have a server farm to run our fuzzer on, instead, we only had 2 machines which ran with Intel i5-3470 processors and hard drives (one was fuzzing XSLT and the other was fuzzing HTML). We were only able to run the fuzzer for a total of 11 days.

Although we have made some improvements to Domato's grammar, there is still room for improvement. Firstly, we could improve the JS, HTML and XSLT grammar with the same workflow used to improve the CSS and SVG components. Secondly, we could also add code coverage analysis into Domato to increase its effectiveness. Also, to actually find more bugs within Firefox, we should not only use our improved Domato fuzzer but also utilise other open source fuzzing projects such as Mozilla's DomFuzz Project.[12] Apart from just finding bugs in Firefox, we can attempt to run the modified fuzzer on other browsers or other platforms (such as mobile).

## 6. Conclusion

We extended the CSS and SVG components and implemented XSLT fuzzing in Domato and fuzzed Firefox for 11 days, finding 3 unique denial-of-service crashes.

**Acknowledgements**

We would like to thank our mentors, Sin Mong Leng and Ho Wei Xiong for their constant guidance and unwavering support. We would also like to thank our schools, Temasek Junior College and Singapore Chinese Girls' School, for giving us the opportunity to do this project.

## References

[1] Berr, J. (2017, May 16). "WannaCry" ransomware attack losses could reach $4 billion. Retrieved from https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/

[2] Desktop internet browser market share 2018 | Statistic. (n.d.). Retrieved from https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/

[3] Chrome Rewards – Application Security – Google. (n.d.). Retrieved December 20, 2018, from https://www.google.com/about/appsecurity/chrome-rewards/index.html

[4] Ivan Fratric. Domato. (last updated, 2018, July 04). Googleprojectzero/domato. Retrieved from https://github.com/googleprojectzero/domato

[5] SkyLined. BugId (accessed 2018, November 21). SkyLined/BugId. Retrieved from https://github.com/SkyLined/BugId

[6] Fuzzing. (n.d.). Retrieved December 20, 2018, from https://www.owasp.org/index.php/Fuzzing

[7] Aki Helin / radamsa. (n.d.). Retrieved December 20, 2018, from https://gitlab.com/akihe/radamsa

[8] Basic zzuf usage. (n.d.). Retrieved December 20, 2018, from http://caca.zoy.org/wiki/zzuf/tutorial1

[9] HTTrack Website Copier - Free Software Offline Browser (GNU GPL). (n.d.). Retrieved from https://www.httrack.com/

[10] Tutorialspoint.com. (n.d.). XSLT Tutorial. Retrieved December 20, 2018, from https://www.tutorialspoint.com/xslt/

[11] HTML. (n.d.). Retrieved from https://www.w3schools.com/

[12] MozillaSecurity. (2018, June 29). MozillaSecurity/domfuzz. Retrieved December 20, 2018, from https://github.com/MozillaSecurity/domfuzz

**Appendix**

1. HTML fuzzing script
2. XSLT fuzzing script
3. Static XML used for testing
4. CSS property extractor python script
5. CSS value extractor python script
6. Triaging Script (phase 1, HTML)
7. Triaging Script (phase 1, XSLT)
8. Triaging Script (phase 2)
9. Minimised Version of ‘Breakpoint Reached’ Code

```
REM HTML Fuzzing Script
:loop

for /F %%f in (data_html.txt) do @set cur=%%f
REM Generate Files
call domato-master\generator.py --output_dir test_files --no_of_files 100

REM Testing the Files
call BugId-master\PageHeap.cmd firefox ON
call BugId-master\PageHeap.cmd "minidump-analyzer.exe" ON
for %%f in (test_files\*.*) do (
    echo Testing File: %%f

    REM Run the Actual Program
    call BugId-master\BugId.cmd -q -f --nApplicationMaxRunTimeInSeconds=30
"C:\Program Files\Mozilla Firefox\firefox.exe" -- %%f

    if ERRORLEVEL 1 (
      echo BUG Detected
      copy %%f buggy_files /Y
    )

)
set /a cur=%cur%+100
echo %cur% > data_html.txt

del test_files\*.* /Q
goto loop
```

```
REM XSLT Fuzzing Script

for /F %%f in (data_xslt.txt) do @set cur=%%f
```

```
:loop
REM Generate Files
call python xslt/xslt_generator.py 100 xslt_files
REM Testing the Files
call BugId-master\PageHeap.cmd firefox ON
call BugId-master\PageHeap.cmd "minidump-analyzer.exe" ON
for %%f in (xslt_files\*.*) do (
    REM Prep the test xml file
    echo ^<?xml version="1.0" encoding="UTF-8"?^> > xslt_cur.xml
    echo ^<?xml-stylesheet type="text/xsl" href="%%f"?^> >> xslt_cur.xml
    type xslt_test_template.xml >> xslt_cur.xml

    echo Testing File: %%f
    REM Run Firefox
    call BugId-master\BugId.cmd -q -f --nApplicationMaxRunTimeInSeconds=30
"C:\Program Files\Mozilla Firefox\firefox.exe" -- xslt_cur.xml

    if ERRORLEVEL 1 (
      echo BUG Detected
      copy %%f buggy_files /Y
    )


)
@set /a cur=%cur%+100
echo %cur% > data_xslt.txt

del xslt_files\*.* /Q
goto loop
```

```
<?xml-stylesheet type="text/xsl" href="xslt_out.xsl"?>-->
<catalog>
```

```
<cd release_date="090202">
    <title>
     <main_title>Empire Burlesque</main_title>
     <subtitle>Fights Back</subtitle>
  </title>
    <artist type="simple">
     <name race="chinese">
     Bob Dylan
     <middle_name type="True">Robert</middle_name>
     </name>
  </artist>
    <country>USA
     <city state="Singapore"/>
     <lane number="10"/>
     <floor value="10">
          <unit> 123 </unit>
     </floor>
  </country>
    <company>Columbia
     <salary position="vice-president">300</salary>
     <location>
          <country>Singapore</country>
          <city>Singapore
               <random>RANDOM</random>
          </city>
     </location>
  </company>
    <price currency="SIN">10.90</price>
    <date day_of_week="Monday">
     <year>1981</year>
     <month>03</month>
```

```
      <day>21</day>
    </date>
  </cd>
```

**Everything in the <cd></cd> is copied many times**

```
</catalog>
```

```
#CSS Property Extractor
import os
import re

def css_extract():
     directory = os.fsencode("cssfiles")
     all_properties = []
     for file in os.listdir(directory):                    #For each file in
cssfiles directory --> filename
     filename = os.fsdecode(file)
     print("Current File:", filename)
     line = ""
     try:
          with open("cssfiles//" + filename) as f:         #puts all chars into
1 line
               for l in f:
               line += l
     except:
          print("File parsing error, skipping file")
          continue
     line = "".join(line.split())                     #Removes all white spaces
from line
     #line = "This{width:30; box-size:10;box-size:200; }"
     properties = re.findall(r"""                     #All strings that match
the regular expression
                                 (?<=(?:               #Positive look back
                                \{|;))               #{ | ; (starting 
characters)
                                ([a-zA-Z\-@]+)           #Captures property 
(must consist of alphebetic chars or - @)
                                (?=:)                    #Positive look ahead 
for : (terminating character)
                                """,line,re.X)

     for css_property in properties:
          if len(css_property) > 2:                        #Removing CSS 
Variables
               if css_property[0] == css_property[1] == '-':
               continue
          all_properties.append(css_property.lower())      #Adds properties to 
all property list, ensures small letters

     print("Starting Processing--------------------------------")
     all_properties = list(set(all_properties))           #Removes duplicated 
properties
     all_properties = sorted(all_properties, key=str.lower)  #Lexicographically 
sorts properties
     fileout = open(r"property_files//css_property_output.txt","w")
     #Writing properties to text file
     for css_property in all_properties:
     #print(css_property)
     fileout.write(css_property+"\n")

     #fileout.write("Total Number of properties: " + str(len(all_properties)))
     fileout.close()

     print("Number of Properties from css files:", len(all_properties))
```

```
def compare_missing_properties():
     domato_filename = r"domato_css_properties.txt"
     property_filename = r"all_properties.txt"
     missing_properties = []

     with open(domato_filename) as df:
     domato_properties = set(["".join(l.split()) for l in df])

     with open(property_filename) as pf:
     properties_to_add = ["".join(l.split()) for l in pf]
     properties_to_add.pop()                          #Removes line about how
many properties there are

     for css_property in properties_to_add:
     if css_property not in domato_properties:
          missing_properties.append(css_property)

     fileout = open(r"missing_properties.txt","w")         #Writing properties
to text file
     for css_property in missing_properties:
     print(css_property)
     fileout.write(css_property+"\n")

     fileout.write("Total Number of properties: " +
str(len(missing_properties)))
     fileout.close()
     print("Number Missing of Properties:", len(missing_properties))



#MAIN
```

```
CODE################################################################################
##################################
css_extract()
directory = os.fsencode("property_files")
all_properties = []
for file in os.listdir(directory):                    #For each file in
cssfiles directory --> filename
     filename = os.fsdecode(file)

     try:
     print("Current Property File:", filename)
     with open("property_files//" + filename) as f:
          for line in f:
               line = "".join(line.split())
               all_properties.append(line)
     except:
     print("ERROR WITH FILE:", filename)
     continue

all_properties = list(set(all_properties))
all_properties = sorted(all_properties, key=str.lower)
fileout = open(r"all_properties.txt","w")        #Writing properties to text
file
for css_property in all_properties:
     #print(css_property)
     fileout.write(css_property+"\n")

fileout.write("Total Number of properties: " + str(len(all_properties)))
fileout.close()

print("Number of Properties:", len(all_properties))
```

```
compare_missing_properties()
```

```
#CSS Value Extractor
import re
import os

def load_files(directory="cssfiles"):
	print("Loading Files...")
	#Compacting CSS files into list of strings
	files = []
	err = 0                                                     #Error counter
	#directory = os.fsencode("cssfiles_test")
	for file in os.listdir(directory):                          #For each file
in cssfiles directory --> filename
		filename = os.fsdecode(file)
		#print("Loading:", filename)
		line = ""
		try:
			with open(directory+"//" + filename) as f:          #puts all
chars into 1 line
				for l in f:
					line += l
		except:
			err += 1
			#print("File parsing error, skipping file")
			continue
		line = "".join(line.split())                        #Removes all white
spaces from line
		files.append(line)
	print(err, "files skipped due to parsing error")
	return files
```

```
###########################################################
files = load_files("cssfiles")                            #directory for css
files
property_file = "missing_properties.txt"                  #file of properties
to check
output_properties_file = "additional_css.txt"             #file to copy paste
to css.txt
output_values_file = "additional_cssproperties.txt"                  #file to
copy paste to cssproperties.txt
###########################################################

counter = 0
property_value_pairs = {}
outp = open(output_properties_file, 'w+')
outv = open(output_values_file, 'w+')
#Finding Values for each property
print("Getting values...")
with open(property_file) as pf:
     for line in pf:
     #Extracting values-----------
     cur_property = "".join(line.split())
     print ("Getting", cur_property)

     all_values = []
     for file in files:
          values = re.findall(r"""                         #All strings that
match the regular expression

                                 (?:\{|;)(?:"""+cur_property+""":)
                                 ([^;\}]+)
                                 (?:\}|;)
```

```
                              """,file,re.X)
            values = list(set(values))
            for value in values:
                  all_values.append(value)

      all_values = tuple(sorted(set(all_values), key=str.lower))
      property_value_pairs[cur_property] = all_values

      #FORMATTING-------------------
      all_values = list(all_values)
      new_values = set()
      if len(all_values) == 0:
            print ("No Values,
Skipping.........................................")
            continue

      counter+= 1
      outp.write("#" + cur_property + '\n')
      outp.write("<new_cssproperty> = "+cur_property+":
<cssproperty_"+cur_property+">\n")
      outp.write("<cssproperty_name>  = "+ cur_property + "\n")
      outp.write("<cssproperty_value> = <cssproperty_" +cur_property +">\n\n")

      outv.write("\n#Values for "+cur_property+"\n")
      for value in all_values:

            #[0-9]*\.?[0-9]* means floating point numnber
            value = re.sub(r'#[0-9a-fA-F]+', r"<color>", value)
            value = re.sub(r'rgba\([0-9]*\.?[0-9]*,[0-9]*\.?[0-9]*,[[0-9]*\.?[0-
9]*,[0-9]*\.?[0-9]*\)', r"<color>", value)
            value = re.sub(r'[0-9]*\.?[0-9]*%', '<percentage>%', value)
            value = re.sub(r'[0-9]*\.?[0-9]+', r"<float>", value)
            value = re.sub(r'[\d]+', r"<fuzzint>", value)
            new_values.add(value)

      for value in new_values:
            outv.write("<cssproperty_"+cur_property+"> = "+value+"\n")

outp.close()
outv.close()

print ("Finished,", counter, "properties added")
```

```
REM HTML Triage Script
call BugId-master\PageHeap.cmd firefox ON
call BugId-master\PageHeap.cmd "minidump-analyzer.exe" ON
for %%f in (buggy_files\*.*) do (
    echo Testing File: %%f

    REM Run the Actual Program
    REM BugId-master\BugId.cmd %WinDir%\system32\rundll32.exe -q -- advapi32
CloseThreadWaitChainSession

    call BugId-master\BugId.cmd -q --bGenerateReportHTML=true "--
sReportFolderPath=\"BugId_report\"" --nApplicationMaxRunTimeInSeconds=30
"C:\Program Files\Mozilla Firefox\firefox.exe" -- %%f
)

call python triage2.py
```

```
REM XSLT Triage Script
call BugId-master\PageHeap.cmd firefox ON
call BugId-master\PageHeap.cmd "minidump-analyzer.exe" ON
```

```
for %%f in (buggy_files\*.xsl) do (
    echo Scanned File: %%f

    REM Run the Actual Program
    REM BugId-master\BugId.cmd %WinDir%\system32\rundll32.exe -q -- advapi32
CloseThreadWaitChainSession

    echo ^<?xml version="1.0" encoding="UTF-8"?^> > %%f.xml
    echo ^<?xml-stylesheet type="text/xsl" href="%%~nxf"?^> >> %%f.xml
    type xslt_test_template.xml >> %%f.xml
)
for %%f in (buggy_files\*xml) do (
    echo testing file %%f
    call BugId-master\BugId.cmd -q --bGenerateReportHTML=true "--
sReportFolderPath=\"BugId_report\"" --nApplicationMaxRunTimeInSeconds=60
"C:\Program Files\Mozilla Firefox\firefox.exe" -- %%f
)

call python triage2.py
```

```
#Triaging Script Phase 2
import re
import os
import shutil

d = {}
for filename in os.listdir('BugId_report'):
      print filename
      key = re.findall(r"^[\w\s]+", filename)
      key = key[0]
      if key in d:
      d[key].append(filename)
```

```
     else:
     d[key] = [filename]

     if not os.path.exists('sorted_report\\' + key.replace(" ", "")):
     os.makedirs('sorted_report\\' + key.replace(" ", ""))
     shutil.copyfile('BugId_report\\' + filename, 'sorted_report\\' +
key.replace(" ", "") + '\\' + filename)

    #copy the actual buggy file
     report = open('BugId_report\\' + filename, 'r')
     text= report.read()
     source_file_name = re.findall(r"(?:<td>Arguments:
<\/td><td>\['buggy_files\\\\)([^']+)(?:'])", text)
     source_file_name = source_file_name[0]
     shutil.copyfile('buggy_files\\' + source_file_name, 'sorted_report\\' +
key.replace(" ", "") + '\\' + source_file_name)
     print (source_file_name + " copied")
     if source_file_name[-4:] == '.xml':
     source_file_name = source_file_name[:-4]
     shutil.copyfile('buggy_files\\' + source_file_name, 'sorted_report\\' +
key.replace(" ", "") + '\\' + source_file_name)
     print(source_file_name + " copied")
     report.close()
```

```
<!-- Minimised Version of ‘Breakpoint Reached’ Code -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html><body><xsl:apply-templates/></body></html>
</xsl:template>

<xsl:template match="/catalog">
<xsl:for-each select="/catalog/cd">
```

```
    <xsl:value-of select="/"/>
    <xsl:value-of select="/"/>
</xsl:for-each>

<xsl:for-each select="/">
    <xsl:apply-templates/>
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```