The Effects of Fuzzing with Unicode Characters on Web Browsers

Tejas Prasad
CSE637S: Software Security
Washington University in St. Louis
St. Louis, Missouri
tprasad@wustl.edu

***Abstract*—The Unicode library is continually updated every year, adding new characters, symbols and languages each time. This means existing software which utilizes the Unicode standard has to be updated as well to handle these new characters and languages. There currently is not a focus on checking whether an application can handle different Unicode characters through the use of a Fuzzer. This opens the possibility of a vulnerability in the software for malicious users. In this project, I will update a fuzzer and a fuzzing library to handle the latest Unicode standard and test the ability of common web browsers to parse edited HTML files to see if there are any security vulnerabilities or bugs when interacting with Unicode characters.**

***Keywords*— Unicode, fuzzing, Internet Explorer, JavaScript, Web Browsers, HTML, Chrome, Firefox**

I. Introduction

The internet is connecting more people internationally as days go by. As such, these people need to communicate in their own language. The problem is that the internet was built on using the English language with many protocols and programming languages using and interpreting English words. As such, the Unicode standard was created as a way to display and use other languages.

Fuzzing is a way of providing random input to various applications to see if that input causes any bugs or vulnerabilities. Almost all fuzzers today use mainly English or ASCII characters to generate the random input in all of their test cases. However, new characters are being used due to the expanding Unicode standard each year. Each new Unicode standard adds support for multiple new languages. Unicode 11, for example, added Mayan numerals, Indic Siyaq numbers, and the Dogri language to name a few. Unicode 12 also for example, was released in March 2019 and added support for the Wancho and Nyiakeng Puachue Hmong languages.

Web browsers are the only software which interacts with characters from different languages on a daily basis. Many other applications today do not expect Unicode characters as an input and due to this, bugs and vulnerabilities within them exist. One example is a bug within Apple's OS. When a unique Telugu Unicode character was provided as an input to any application in the OS such as the messaging or email app, the system crashed. This happened because the Unicode character caused a buffer overflow and the operating system booted into its safe mode in an attempt to prevent further damage. This buffer overflow happened because the one character provided as an input was actually comprised of multiple Unicode characters. This bug was prevalent on all of Apple's operating systems and would crash applications from messaging, email, to the network app.

Adding Unicode characters to generate test cases with a Fuzzer is something which is not currently done and could allow for unintended outputs for an application as demonstrated in Apple's OS. By adding these characters, developers can see if their software can handle Unicode characters and help to prevent bugs such as a buffer overflow which can potentially become a vulnerability for malicious users to exploit.

II. Existing work

A. *Unicode Library*

A Unicode library for fuzzing exists which was developed in 2016 [1]. This library however only looks for Unicode characters which are equivalent to a single character. Additionally, this library is outdated. The current Unicode standard is Unicode 11. The library was created using a small subset of characters from Unicode 6 and 9 and is created entirely in C# and XML.

B. *Unicode Testing*

As it stands, no papers were published which utilized Unicode fuzzing specifically. The only reference is a presentation by the developer who made the fuzzing library. This presentation goes over how Unicode characters can be used to spoof URLs, directory names, and file names for example [2]. The presentation also discusses the different attack vectors and root causes of Unicode fuzzing. No mention is made of a specific fuzzer the author utilized.

C. *Browser Fuzzing*

A browser fuzzer exists named BFuzz which utilizes an HTML fuzzer called Domato. The HTML fuzzer can create html test cases with CSS, HTML, or JavaScript. Domato has already been used to find bugs in Safari, Chrome, Firefox, Internet Explorer, and Microsoft Edge [4]. BFuzz then takes these test html files and opens them in either Firefox or Chrome to see if they run or crash [3]. This fuzzer is made entirely in Python and has been used to find bugs in Firefox.

III. Methodology

Initially, the first design choice was to decide which Unicode characters to include. The characters from the language of Telugu was an easy decision as these characters were used to cause a bug within Apple's suite of operating systems. It is also a largely used language which has a strong presence on the internet. I decided to use the language of Dogra from the Unicode 11 standard as it was newly added and was unlikely to have a large amount of people using the language daily on the internet. While working on this project, the Unicode 12 standard was released. I took this as an opportunity to utilize a new language which has not been supported on web browsers before. As such, I decided the Nyiakeng Puachue Hmong language. Since the Unicode standard does not cover languages only, I decided to utilize symbols and emojis within the project as well. Symbols such as ballet shoes, a yo-yo, a chair, a razor, and others were chosen.

The existing Unicode library was originally created with characters from the Unicode 9 standard. These characters were stored in an XML file. The new characters from Unicode 11 and 12 were also saved in an XML file and the same format was kept so no additional changes would be required to have the library parse through the file.

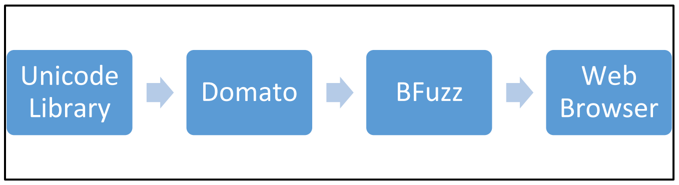

Fig. 1. Domato would generate HTML files with Unicode characters it got from the Uncide Library. BFuzz would then take these genreated files and open them in a specific web browser.

As shown in Figure 1, the initial idea for the project was to build the Unicode library as a DLL file and link it in accordingly to the HTML fuzzer, Domato. The HTML fuzzer would then utilize the function calls available to it to go through a file and randomly change an ASCII character to a Unicode character which either does not match the ASCII value or is the best fit match for the value. Domato would use this idea to create as many HTML files as necessary. These files would then be passed into the browser fuzzer, BFuzz. This fuzzer would open each file in an instance of a selected web browser.

There were issues with this methodology. One issue was prior to testing, a user would have to run Domato to generate test cases and then run BFuzz to use those generated test cases which required editing the Python file of BFuzz to point to a new directory every time if the user wanted to create a new directory of test cases. Additionally, this method had no logging implemented anywhere. So, a user would not know what exact ASCII characters were replaced with Unicode characters. The user would also be required to record if a HTML file lead to a crash or if it loaded correctly and what occurred.

While developing the library, I found DLL files compiled from C# are not the same as DLL files compiled from C. As such, the built-in Python library of ctypes could not be used. I attempted to use additional C# and Python libraries to connect the Python fuzzer with the C# library which all failed to allow Python to access the functions available within the library. In the end, I decided to rewrite the library in Python as this would remove the dependence on knowing and using two programming languages.

The updated methodology for the project was to integrate the Unicode library, the browser fuzzer, and the HTML fuzzer together so only a single program would have to run, and everything would be automated. The user would start up the browser fuzzer, specify how many HTML files to generate per round of testing, and how many rounds of testing to do. This one program would also monitor the file currently running to record its CPU usage, memory usage, and whether the file cause the browser to crash.

The grammar file for the HTML section and the JavaScript section was not edited as the purpose of this project was to test if only adding a Unicode character would result in new or different bugs.

IV. Developed Code

A. Unicode Library

```
def getUChar():
    unicodeList = []
    for unicode in ucharRoot.findall('Mapping'):
        string = '\\u' + unicode.find('Unicode').text

        unicodeText = string.encode('utf-8')
        decodedValue = unicodeText.decode('unicode_escape')
        unicodeList.append(decodedValue)

    if len(unicodeList) > 0:
        randIndex = random.randint(0, len(unicodeList) - 1)
        return unicodeList[randIndex]
    else:
        return '\\u0263'.encode('utf-8').decode('unicode_escape')
```

Fig. 2. This is the Python code to parse through the XML file, encode it into a value which Python is able to write to a file, add the value to a list, and randomly grab a value to return.

A new XML file was created to hold the new language, Dogra, which was added in Unicode 11 and the new language, Nyiakeng Puachue Hmong which was added in Unicode 12. Additionally, the characters for Telugu and various symbols were added as well. Through the use of the Python XML library, the XML file would be imported and generated as a tree. This would allow for easy and quick traversal through the file. For the newly added characters, the code in Figure 2 was used. For the existing library, a different function was created. This function would take in a character, turn it into its ASCII value, and find the Unicode character which matches the ASCII value.

This should in theory allow for the original character to be swapped out with a Unicode character with no issues.

B. HTML Fuzzing

The HTML fuzzer, Domato, was edited to utilize the new Unicode library redeveloped in Python. Once I got into editing fuzzer, it became apparent to me the previous idea of going through the completed HTML file and randomly changing characters to Unicode characters was not a viable solution. The completed file had three sections, a CSS, JavaScript, and HTML section. Going through the CSS and JavaScript sections randomly and changing characters would cause a syntax error on the file and the web browser would not be able to properly load it. As such, I decided to separately fuzz the JavaScript and HTML sections. The CSS section was left out as the chances of finding a bug from the introduction of Unicode characters to it was low. A set of new functions were created to randomly add in Unicode characters to only the HTML section of the HTML file.

```
def fuzzHTML_File(file, logger):
        size = len(file)
        resultList = list(file)
        i = 0
        for character in resultList:
            randInt = random.randint(0, 100)
            if randInt > 0 and randInt < 24:
                resultList[i] = getBestFit(character)
                logger.debug('HTML_Fuzz: At %d: Char: %s', i,
resultList[i].encode('utf-8'))
            elif randInt > 25 and randInt < 35:
                resultList[i] = getExpandedUnicode()
                logger.debug('HTML_Fuzz: At %d: Char: %s', i,
resultList[i].encode('utf-8'))
            elif randInt > 55 and randInt < 75:
                resultList[i] = getMalformBytes(character)
             else:
                resultList[i] = getUChar()
            i = i + 1

        temp = ''.join(resultList)

        index = random.randint(0, size - 1)
        chance = random.randint(0, 100)

        if chance > 25 and chance < 30:
            temp = insertJoinerUnicode(temp, index, logger)
        elif chance > 5 and chance < 9:
            temp = insertPrivateUseAreaUnicode(temp, index,
logger)
        elif chance > 35 and chance < 40:
            temp = insertRightLeftReadingUnicode(temp, index,
logger)
        elif chance > 78 and chance < 87:
            temp = insertVowelSepUnicode(temp, index, logger)

        return temp
```

Fig. 3. The Python code which iterates through the HTML section of a file and fuzz it.

As shown in Figure 3, the function randomly changes a character with some probability by either adding in a completely random Unicode character, a Unicode character which is considered the best fit, or by editing the bytes of the character by slicing off the last byte. This getMalformBytes function was used by the creator of the Unicode library and was ported over when the library was converted to Python. The probabilities shown above were chosen arbitrarily.

A set of invisible characters were also randomly added into the section. These characters were added in an attempt to induce a bug similar to the one found in Apple operating systems since that error was due to one character actually being made up of multiple unicode characters.

Additionally, a logger was used to record which character was added so any user can look through the log to see if adding a specific Unicode character lead to unintended results.

C. JavaScript Fuzzing

The challenge of trying to fuzz JavaScript code is trying to make sure the code still compiles. The solution I found to this issue is by going through the section of code and renaming a variable. Essentially refactoring the variable. Figuring out what the variable will be named was simple as the generated file has a standard variable name starting from “var00001” and going till however many variables a file generates.

```
def fuzzJavaScript(code, logger):
    variableName = ' var000'
    notFoundCount = 0
    newCode = code
    for i in range(1, 100):
        variable = variableName
        if i < 10:
            variable = variableName + "0" + str(i)
        else:
            variable = variableName + str(i)

        if code.find(variable) != -1:
            notFoundCount = 0
            prob = random.randint(0, 100)

            if prob >= 0 and prob <= 10:#replace word with
20% chance
                newWord = generateUnicodeWord()

                logger.info('JavaScriptFuzz: Replacing: %s With:
%s', variable, newWord.encode('utf-8'))
                newCode = newCode.replace(variable,
newWord)

        else:
            notFoundCount = notFoundCount + 1

        if notFoundCount >= 3:
            break

    return newCode
```

Fig. 4. Python code to iterate through the JavaScript section of a file and fuzz it.

Through the use of the Python string library, I looked for the strings “var00001” till “var00100”, exiting this loop if I could not find 3 variables in a row. If a variable name was found, the function would then calculate a probability of replacing the variable with a new one. The new variable was created by a new function. This function would create a new variable name comprised of a mix of Unicode characters and ASCII characters based on an arbitrary probability and would have a random length between 4 and 15 letters.

D. Browser Fuzzing

The BFuzz Python program was edited to integrate the HTML Fuzzer, Domato, so only BFuzz would have to run and not Domato and then BFuzz. The initial prompt was edited to allow the user to specify the number of files to generate per round of testing and how many rounds of testing to run for. Additionally, the web browser Internet Explorer was added in as a choice to run the tests on.

The original BFuzz opened all of the HTML files in new tabs in a single instance of a web browser. I edited the program to kill the browser after a predetermined amount of time and launch a new instance with the next HTML file.

To try and improve the fuzzer, I attempted to automate the testing by having the fuzzer record for each file the CPU usage, memory usage, and whether the file caused the browser to crash or not launch. These statistics would be recorded through the use of the psutil Python library and by tracking a process through its process ID. I discovered here that launching the web browser with a specific file causes the process which launched it to not exist after the browser has started. Additionally, there are multiple processes per browser which run and all have of them have the same executable name. This prevented me from implementing logging on the browser for CPU and memory usage and whether the file caused a crash or could not be opened.

This testing was done with version 73.0.3683.86 of Google Chrome, 11.437.17763.0 of Internet Explorer, and 66.0.2 of Mozilla Firefox. All three versions are for 64-bit. The Python programs were compiled using 64-bit Python 3.6.6.

V. Results

A. Original Implementation

To develop a baseline, the original Domato program and BFuzz program were run. These programs did not have any Unicode characters.

A test of 50 files were generated and opened with BFuzz. The baseline establishes that the HTML files result in minimal amounts of HTML code or characters to be printed on the screen. Many of the pages exhibit animation, forcing the browser to go to full screen, or various shapes.

B. Updated BFuzz with only HTML Fuzzing

The fuzzer ran and generated 10 new files for each round for five rounds. When the fuzzer was run and only the HTML section was fuzzed, every file could be opened on any of the web browsers. The resulting pages ended up in one of three states. Either the page would be completely blank with a solid color on it, the page would have portions of HTML code on it, various shapes along with HTML code, or the page would have a combination of the three previous states.

C. Updated BFuzz with HTML and JavaScript Fuzzing

For this test, the fuzzer also generated 10 new files for every round for five rounds. When the fuzzer run and the HTML and JavaScript sections were fuzzed, every file could be opened on any of the web browsers.

There is an interesting difference between this test and the HTML only test. When the JavaScript section is also fuzzed, it leads to the page to also print out characters. When these files are opened in Internet Explorer, the browser is not able to figure out what characters the HTML file has. When these files are opened in either Google Chrome or Mozilla Firefox, the browser is able to resolve what the character is and displays it appropriately.

One result which is also interesting is fuzz-6.html. This file can be opened and runs normally on Firefox and Google Chrome. However, the file causes Internet Explorer to freeze up and stop responding. I tested the file with Microsoft Edge to test if the issue occurred on other browsers made by the same company and the file loads normally there. One possibility as to why the file does not load on Internet Explorer, is that the text within the file when it is opened increases exponentially in size and has multiple lines of characters. These two situations together could cause the browser to crash.

VI. Observations

A. Unicode Characters

The first conclusion I can draw based on the testing so far, is that the addition of Unicode characters does not lead to a file to fail to run on any of the tested browsers. The one case where a file does not load, the case is more likely due to exponential increases in text size and multiple lines of characters than the singular addition of Unicode characters.

Additionally, it appears that the addition of Unicode characters leads to the animations of a page to break. This could be a symptom of creating and using a dumb fuzzer to fuzz HTML files. This issue may go away with a smarter fuzzer which doesn’t rely on arbitrary and random probabilities to change a character.

B. HTML Fuzzing

Fuzzing the HTML section of a web browser leads to few if not zero significant results. The HTML parser for web browsers are well tested and created to handle characters from multiple languages. As such, it was not the best target to pick to fuzz with Unicode characters.

C. JavaScript Fuzzing

Fuzzing the JavaScript section of a web browser leads to less varied results than solely with HTML fuzzing. The fact that characters appear on the page more often shows that fuzzing JavaScript leads to consistent behavior. The interesting part is how the characters displayed do not resemble HTML code and if they do, only a few fragments are similar enough. More testing needs to be done with different probabilities of fuzzing a character or variable.

VII. Running the Project

Running the project is a simple process. To run the fully automated project, run the Python script BFuzz.py and follow the prompts on the screen.

To only generate HTML files, run Domato by running the generator.py script with either a single argument of the name of the file you want to generate, or two arguments of the output directory and the number of files.

```
python generator.py <output file>

python generator.py --output_dir <out directory> --no_of_files <number of output files>
```

Fig. 5. The arguments for running Domato to generate HTML files.

To run the non-automated version of BFuzz, run the program BFuzz_Standalone.py and follow the prompts on screen. You will need to select a browser to run on, a timeout, and the directory to look at.

References

[1] Weber, C. (2016). cweb/unicode-hax. [online] GitHub. Available: https://github.com/cweb/unicode-hax.

[2] C. Weber, “Unicode Transformations: Finding Elusive Vulnerabilities,” (2009). Available:https://www.owasp.org/images/5/5a/Unicode_Transformations_Finding_Elusive_Vulnerabilities-Chris_Weber.pdf.

[3] “RootUp/BFuzz”, GitHub. [Online]. Available: https://github.com/RootUp/BFuzz.

[4] I. Fratric, "googleprojectzero/domato", GitHub, 2017. [Online]. Available: https://github.com/googleprojectzero/domato.

[5] C. Webber, “Unicode-Hax,” https://github.com/cweb/unicode-hax.