Using Code Coverage to optimise Web Browser Fuzzing

James Fell
Department of Computer Science
University of York
York, United Kingdom
james.fell@alumni.york.ac.uk

***Abstract*—Identifying vulnerabilities in software products has been an important area of security research for many years. One of the most successful approaches to this problem so far has been that of fuzz testing or fuzzing. This process involves repeatedly passing invalid inputs into an application at runtime and monitoring it with a debugger or other instrumentation for evidence of potential vulnerabilities being triggered, such as observing a segmentation fault or memory corruption. One of the main ways of measuring how thoroughly the fuzzing process has tested a given piece of software is by measuring code coverage throughout the process. If part of the target program has not been executed during fuzzing then any vulnerabilities in that part obviously cannot be detected. This paper presents the development of a suite of software tools that enable the fuzzing of a web browser such that the code coverage obtained during the testing is increased compared to traditional approaches. The prototype system is evaluated through a number of experiments and is finally used to fuzz a selection of web browsers. Multiple previously unknown bugs are detected in several different web browsers including Edge, Internet Explorer and Firefox, some of which appear to be exploitable security vulnerabilities.**

Keywords—fuzzing; code coverage; zero day vulnerabilities; dynamic analysis; web browser security.

I. Introduction

Techniques for identifying software vulnerabilities can be divided firstly into two different approaches; static analysis and dynamic analysis. Static analysis of software involves ways of examining the source code or compiled binary without executing it. Dynamic analysis involves examining the software at runtime, typically after attaching some kind of debugger. Both of these approaches have their relative advantages and disadvantages [12].

For dynamic analysis the most common automated technique for finding vulnerabilities is the process of fuzz testing or fuzzing. In essence this consists of repeatedly giving an application invalid input and monitoring for any sign of this triggering a bug, such as the application crashing or hanging [10]. There are many advantages to this approach such as the ease of automation and the ability to test even very large applications where code review would be too time consuming. It is also the case that each bug discovered by fuzzing automatically comes with its own 'proof of concept' test case proving that the bug can definitely be triggered by a user. In practice many serious and high profile memory corruption vulnerabilities today are discovered through fuzzing.

In order to fully test a web browser, or any other application, using fuzzing, it is important to somehow ensure that the combination of test cases being used executes as large a proportion of the application as possible. That is to say, it is necessary to maximise code coverage [13]. If large sections of code are not executed during the fuzzing process, it is certain that any vulnerabilities that do exist in those areas will not be found. It is this problem that we are concerned with in this paper.

The research presented in this paper firstly contributes a novel method of creating HTML corpora for use in the mutation fuzzing of web browsers. This method is empirically proven to result in superior code coverage compared to corpora made by collecting samples of real world HTML web pages. The same method is then applied to DOM fuzzing of web browsers with similar coverage gains, and some new methods for facilitating crash reproduction are presented.

This paper is organised as follows: Section II introduces fuzzing. In Section III, we discuss the motivation for our work. The proposed approach and the development of our new fuzzing tools is described in Section IV. Section V and Section VI present respectively results to date and future work. Finally, conclusions are given in Section VII.

II. Fuzzing

A. *Origins of Fuzzing*

The term fuzzing was first used in 1988 by Professor Barton Miller at the University of Wisconsin. Whilst connected to a University terminal via a dial-up connection during a storm, Miller noticed that line noise was causing extra, random characters to be added to the commands that he was sending [2]. In addition to the obvious usability problems that one would expect from this, it was observed that the corrupted inputs often caused the various UNIX utilities that he was using to crash and 'core dump'.

This experience motivated Miller and other researchers at the University to create a tool for generating random strings and then pipe its output into the input of a range of UNIX utilities. The group tested almost ninety different UNIX utilities this way on seven different UNIX distributions and found that they were able to crash over 24% of them. These experiments were described in detail in [2] and gave birth to a relatively simple way to supplement the formal verification and software testing techniques that already existed.

B. *Instrumentation*

Fuzzing cannot be successful without some form of instrumentation being applied to the target application. At the very least, there must be some way of detecting when the program crashes and logging which test case caused the crash. However, far more than this can be achieved using more advanced instrumentation. It is possible to detect subtle forms of memory corruption and other errors even when they do not cause the program to crash outright [11]. This allows the detection of vulnerabilities that would otherwise be missed. It is also possible to measure code coverage and therefore know which parts of the target application have been tested so far and which have not [13]. This can be simply for informational purposes for the user or can actually be acted upon by the fuzzer itself when creating new test cases.

C. *Code Coverage*

As already mentioned, code coverage refers to tracking which parts of a program have been executed. This is a very important type of instrumentation to have during a fuzzing session as it makes it possible to see how much of the target has actually been tested. Vulnerabilities cannot be found in a section of a program if that section does not get executed by any of the test cases.

In [19] research was carried out showing that measuring the code coverage of an initial corpus, and adding new test cases to it until no further coverage can be gained, is highly beneficial and increases the number of vulnerabilities discovered. This was confirmed in [20]. Further, specific to web browser fuzzing it was documented in both [9] and [22] that code coverage for an initial web browser test corpus can be increased by including existing conformance test suites from the browser vendors as these are designed to test as many parts of the browser as possible.

D. *Synthesis of Test Cases*

There are three main approaches to creating the test cases or inputs for use in a fuzzing campaign; mutation, generation and evolution [13, 21].

Mutation fuzzing works by starting with a corpus of one or more valid input samples. For example, when fuzzing an image viewer one might start with a number of valid JPEG files. These are then repeatedly modified or corrupted to produce new test cases that are each given as input to the software being tested. This is sometimes referred to as 'dumb fuzzing' as the fuzzer has no specific knowledge of the program being fuzzed or the correct format or syntax to use for inputs.

There are many ways in which test cases can be mutated. Walking bit flipping and byte flipping involves inverting sequences of bits at different places in the test case [15]. Deleting segments of the test case and also splicing in sections from other test cases are common mutation approaches [14]. Another approach is to select sections of the test case and repeat them by inserting copies at random positions [22]. A recent example of a mutation fuzzer is Zulu from NCC Group [15].

Generation fuzzing works by creating test cases based only on some kind of model that describes a valid input, such as a grammar of a programming language or a specification of a binary format. Continuing with the image viewer example, with a specification for the PNG image format it would be possible to repeatedly generate slightly invalid PNG files to use as test cases without needing to use any samples of existing files. This is sometimes referred to as 'smart fuzzing' as the fuzzer is applying detailed knowledge of the format or protocol being fuzzed. This generally results in better code coverage and deeper testing of the target program [1]. An example of a generation fuzzing framework is Sulley [16].

Evolutionary fuzzing creates new test cases based on the response of the target program to previous test cases [23, 24]. This can be an extension of either mutation or generation fuzzing, as either technique can be guided to create the new test cases. In either case, the difference with evolutionary fuzzing is the way in which test cases are created based on feedback from the instrumentation so that they evolve towards a specific goal [21]. This goal could for example be the general increase of code coverage, or it could be to evolve test cases that cause the execution of a specific function within the target application.

E. *Fuzzing Web Browsers*

When fuzzing web browsers to find vulnerabilities it is common to focus on the rendering engine [3, 6]. This part of the browser is responsible for taking the HTML documents, CSS and images that make up a web page and turning them into what is displayed to the user. The rendering engine parses HTML and CSS and then constructs a tree structure from this using the DOM (Document Object Model) API. Scripting languages such as JavaScript can then interact with the rendered document through DOM API calls to make changes to it on the client side without further connections to the web server being necessary. Popular rendering engines in use today include EdgeHTML, Trident, Gecko, WebKit and Blink (itself a fork of WebKit) [4]. The rendering engine is a common location to find exploitable UAF (Use-after-free) vulnerabilities [5].

There are two main approaches to fuzzing a web browser's rendering engine; file format fuzzing and DOM fuzzing. The file format fuzzing approach consists of simply applying the mutation or generation techniques that we have already examined to create malformed web pages and providing these to the browser as input. This is therefore applying a generic fuzzing approach and using the same methods that one might use to fuzz a PDF reader or a video player.

The DOM fuzzing approach applies techniques that are specific to web browsers [3, 7] and the actions are carried out by the browser itself through JavaScript. There are many variations but a typical approach is to crawl the DOM tree of a test case document and collect element references. The elements' attributes can then be mutated and random DOM nodes can also deleted. It is possible to perform such mutations on collections rather than individual DOM elements. The DOM tree can be rearranged in various pseudo-random ways. Garbage collection is then triggered and the process is repeated [7, 8]. This can result in the discovery and triggering of various memory corruption bugs.

Creating logical views from a document tree can also be done as part of DOM fuzzing as it forces the rendering engine to attempt to maintain consistency between the logical views and the underlying DOM tree as the mutations occur. This activity can also result in memory corruption bugs being triggered and revealed [7].

DOM fuzzing tools in general have a problem with producing test cases to efficiently reproduce crashes when they are discovered. A typical DOM fuzzing tool builds up the DOM tree by randomly selecting HTML tags from an array and calling the document.createElement method with each. The built-in PRNG (Pseudo-Random Number Generator) in JavaScript (the Math.random function) is often used which cannot be explicitly seeded with a known value. The script then does some mutations to the DOM tree (again often involving the use of Math.random) which may cause a crash, and if the browser does not crash the script reloads the page and starts over. This continues for perhaps several hours until eventually the browser crashes. When the browser does crash there is not a specific test case available with which to reproduce the crash.

Nothing can be written to disk from JavaScript within a web page for obvious security reasons. Most DOM fuzzers instead use the console.log() method from JavaScript to log messages to the browser's console [34]. This information can then be used to try to investigate what happened and reconstruct a test case. This is far from ideal compared to traditional fuzzing where any time a crash occurs the specific test case that caused it is available right away. The consequence of this has been that DOM fuzzers have been good at triggering bugs in web browsers but investigating the bug to determine the root cause and fix it is often challenging.

III. Motivation

As explained in the previous section, existing approaches to fuzzing web browsers have not typically emphasised code coverage. In order to increase the success of any attempt at discovering fresh, exploitable web browser vulnerabilities, it is necessary to look at how greater proportions of the target browser’s code base can be tested.

In [1] experiments were performed to compare the code coverage obtained from mutation and generation fuzzing in the context of targeting a PNG image file reader. It was found that several parts of the PNG format's specification were not in common use and therefore not present in any of the PNG files downloaded from the web. This means that significant parts of a typical PNG image viewer would never be executed when loading readily available images. The only way that the missing pieces of the PNG format would appear in the initial corpus and thus these functions could be tested during fuzzing would therefore be by using a generation fuzzer. The authors concluded that “applications often contain large sections of code that will only execute with uncommon inputs”. In this situation the experiments found that generating test cases resulted in 76% more code coverage than mutation of PNG files downloaded from the Internet [1].

No academic literature was found researching if the experimental results seen in [1] hold true for HTML files with respect to code coverage in web browsers. It may be that the common practice of downloading a selection of random web pages from the Internet to form an initial corpus for browser fuzzing [19] is actually missing out some valid HTML tags and attributes and therefore not getting the best code coverage possible. Researching this issue was one of the aims of the work presented in this paper.

It was also an aim of the work to design a DOM fuzzing tool that overcomes the traditional shortcomings relating to crash reproduction that were explained in Section II.

IV. Proposed Approach

This section presents the approach that was proposed and then implemented in order to emphasise and optimise code coverage during web browser fuzzing. Methods of dealing with crash reproduction problems when DOM fuzzing are also presented. A suite of new fuzzing tools that resulted from this work is also introduced.

A. *Specification of HTML*

We proposed to start by creating the most thorough and complete specification of HTML that we could, including as many unique tags and attributes as possible.

This is necessary in order to be able to identify which parts of the markup language are absent from a given HTML corpus, as well as in order to later know what should appear in an ideal corpus as we generate it. As a browser's rendering engine will contain separate functionality for handling each HTML tag and attribute that it parses it is necessary to make full use of all available HTML tags when creating a corpus in order to get better code coverage. A web page that contains 25 different HTML tags can be expected to result in more code coverage when parsed by a web browser than one containing only 5 of those HTML tags.

As of 28th October 2014 the current version of the language is HTML 5.0 [25] as published by the W3C (World Wide Web Consortium). An official list of all the HTML5 tags and their attributes is available online [26]. This can be used as the starting point for our specification.

It is not enough to only rely on the HTML 5 specification though. As the HTML language has evolved since its beginnings in the early 1990s various tags have been removed from the specification. Many web browsers have not removed the corresponding rendering functionality that handles these obsolete HTML tags though. It is therefore possible that vulnerabilities exist in some browsers in functions that deal with old HTML tags that no longer appear in the official specification. In order to maximise code coverage and increase the effectiveness of a fuzzing campaign it is therefore important that all these old tags are present in the corpus.

Research into this topic online turned up a number of sources of information about obsolete HTML tags [29, 30, 31]. As well as old tags that have been removed from the specification this also revealed the existence of proprietary tags created by Microsoft and Netscape that were actually never in the official HTML specification.

This research resulted in the identification of the following 44 HTML tags that are not in the HTML 5 specification but which nonetheless can be expected to trigger unique code paths in at least some web browsers: *acronym, app, applet, basefont, bgsound, big, blink, center, command, comment, details, dialog, dir, frame, frameset, hgroup, hp0, hp1, hp2, hp3, hype, ilayer, image, isindex, key, layer, listing, marquee, menu, menuitem, multicol, nextid, nobr, noembed, noframes, plaintext, server, sound, spacer, strike, summary, tt, typewriter, xmp*.

An ideal HTML corpus would therefore contain all 44 of those obsolete tags as well as the 108 tags that appear in the HTML 5 specification. In addition to this a total of 147 HTML attributes were identified from both the HTML 5 specification and the available details of the obsolete tags.

We therefore have a list of 152 HTML tags and a list of 147 HTML attributes that can be used as a HTML specification for evaluating a corpus. If any of these tags or attributes are missing from a HTML corpus this should be flagged up as a deficiency that will result in reduced code coverage.

B. *Evaluating Existing Corpora*

Based on the HTML specification that we created, a Python tool called HTMLscan was developed. The purpose of this tool is to evaluate a corpus of HTML files that the user is intending to use for fuzzing, and report on missing HTML tags and attributes. This provides the user a simple way of telling how complete the corpus is and how it can be improved before the fuzzing begins.

Given an existing corpus of HTML files that we wish to evaluate against our HTML specification, it is necessary to first read each file and parse it.

After opening each web page file and reading its content, it is necessary for HTMLscan to carry out the process of lexing or lexical analysis. This involves using regular expressions to group individual ASCII characters together into discrete tokens [35]. Once each token has been identified and classified, the process of parsing can be carried out. This builds up a parse tree data structure based on the specific tokens that appear in the input and their order [36]. Once this has been done we are no longer simply looking at the content of a text file, but rather a data structure that can be iterated over with the HTML tags and attributes clearly identified as such.

After reading and parsing each file in the corpus HTMLscan can build a list of all the HTML tags and attributes that are present in it. It is then a fairly simple matter for the tool to see which HTML tags and attributes appear in our specification lists but do not appear in our lists built from parsing the corpus. These of course are the HTML tags and attributes that are missing from the corpus. These missing HTML tags and attributes are reported to the user and it is recommended that the corpus be expanded by adding this missing HTML. Extra functionality in the target web browser can thus be executed and the probability of discovering vulnerabilities is increased.

C. *Generating Better Corpora*

A Python tool called HTMLgen was next developed with the purpose of generating corpora of HTML pages that make full use of all of the tags and attributes in the HTML specification that we had created. Such corpora would be ideal for mutation fuzzing of web browsers.

Two separate approaches were considered for generating new HTML files from the specification. The simpler option was to take an approach similar to Mangleme [17] where all the HTML tags and their attributes are stored in a two dimensional array and then picked from it in a random order. We could therefore simply select tags and place them in a random order within the new web pages. This is relatively easy to implement but will not generate web pages that are syntactically correct. This is because in HTML it is not valid to just use tags in absolutely any order [25]. As a simple example, <*head*> should be used in the page before <*body*> is and not the other way round. Another example is that you would expect to see an <*input*> tag somewhere between <*form*> and </*form*> rather than just at any random place in the page.

There was, however, an argument in favour of using this method. Rendering engines within web browsers are not strict about HTML correctness and will do their best to render anything even close to correct [28]. This is due to the fact that many different web browsers exist and many different tools for creating web sites exist (as well as people creating them manually by writing their own HTML in a text editor). If web browsers were designed to throw an error and refuse to render anything that did not conform exactly to the HTML standard, it is fair to say that many web sites would not be accessible. Therefore even if the web pages generated by HTMLgen had tags appearing out of order, all major web browsers would still try to render them rather than simply refusing to process them.

The second approach that was considered is to formulate and use a grammar to generate HTML that is syntactically correct and valid. A context-free grammar (CFG) [18] for HTML 2.0 in Backus–Naur Form (BNF) was found to be available online [27]. This could be used as a starting point and be adapted and updated by appropriately adding in production rules for the other tags and attributes from our specification. By incorporating a grammar into the HTMLgen tool and following its rules we can be sure to make use of tags in the correct order and hence generate new web pages that are valid HTML.

Even though the first approach would produce HTML that the browser's rendering engine would attempt to process the second approach is clearly better for the following reason. It is true that whichever mutation fuzzer is used will generate malformed HTML from the corpus by the very nature of fuzzing. However, the initial corpus that HTMLgen generates should still be made up of samples of valid input rather than HTML that is already malformed and incorrect. The mutation engine that is applied to the final corpus should be entirely responsible for the introduction of incorrect syntax.

We therefore chose to use the more challenging of the two possible approaches and construct our own grammar that includes all of the HTML tags and attributes that appear in our specification. The HTMLgen tool uses this grammar and follows its production rules, selecting pseudo-random branches whenever multiple options are available for expanding a non-terminal symbol. This allows the generation of rich, varied corpora that make use of the full range of available HTML tags and attributes. Such corpora can be expected to be ideal for mutation fuzzing and to result in excellent code coverage.

This method differs from the work done by Charlie Miller in [1] regarding fuzzing using PNG files. In that work, after it was seen that PNG files harvested from the web were incomplete compared to the PNG specification, mutation fuzzing was abandoned in favour of generation fuzzing based on the official PNG specification. The approach that we applied to HTML test cases instead consists of using generation to create a syntactically valid corpus before proceeding with mutation fuzzing.

This approach has two advantages. Firstly, by combining both generation and mutation we get the strengths of both. We can generate an initial corpus of valid test cases that makes full use of the HTML specification and gives superior code coverage while still having the simplicity of then applying a mutation engine to carry out the actual fuzzing, including using the generated corpus to seed a third party mutation fuzzer such as Radamsa [32].

Secondly, by our generation tool using not just the latest, official HTML 5 specification but also the old, undocumented and proprietary HTML tags uncovered by our research we benefit from combining the strengths of generating a corpus from a specification with the strengths of harvesting one from the web. The problem with generating a corpus based on only the official, documented specification is that certain undocumented features may be missed out. This results in missed opportunities and less code coverage.

As a historic example of this principle, in [19] it was documented that crawling the Internet to build a corpus of HTTP responses from public web servers resulted in the discovery of an undocumented HTTP response header coming from a particular IIS server. When fuzzing various web browsers by mutating the HTTP protocol it was seen that including this undocumented header in the corpus increased code coverage in the Internet Explorer browser. In turn, mutating this header led to the discovery of a UAF vulnerability in Internet Explorer. This vulnerability had been missed by previous fuzzing campaigns simply because the HTTP response header in question was undocumented and so did not appear in any generated corpus. Only when web crawling was used to collect samples of HTTP responses from the wild was it discovered. Our approach applies this lesson by making an effort to research undocumented, rare and obsolete HTML tags and including these in our generation engine.

When HTMLgen is executed by a user, a command line argument is used to specify how many separate HTML pages the tool should generate. The tool then uses its Context-Free Grammar to generate the requested number of syntactically valid HTML pages and saves these to disk. The user can then use the resulting corpus as the starting point for a fuzzing session.

D. DOM Fuzzing

A Python tool called HTMLfuzz was developed. The purpose of this tool is to apply DOM fuzzing strategies to the same high quality HTML test cases that we can now generate with our custom Context-Free Grammar.

The HTMLfuzz tool binds a HTTP server to a TCP port on localhost for the browser to connect to, then serves up randomly generated HTML test cases along with a fixed JavaScript. The web browser parses the HTML test case and builds the DOM tree, then executes the JavaScript which manipulates the DOM tree in ways designed to trigger bugs. The final action that the JavaScript takes is to reload the page which causes the next test case to be generated. This process therefore continues until either it is terminated by the user, or the web browser crashes.

As explained in Section II there has been a long term issue with DOM fuzzing regarding reproducing crashes when they occur [34]. The HTMLfuzz tool represents a step forward over previous DOM fuzzers in terms of reproducibility of crashes.

When a traditional DOM fuzzer has been running for a period of time and the browser eventually crashes, no unique test case is available to trigger the exact same browser state that led to the crash. All that is available is any information about the fuzzer's state that was logged to the browser's console, which is less than ideal.

When implementing the HTMLfuzz tool, thought was given to how to improve this situation. Because we are generating the HTML test cases from a grammar in our Python, instead of picking random HTML tags using JavaScript within the browser process, it is of course possible to write the current HTML test case to disk immediately before serving it to the web browser.

Each time HTMLfuzz is started up, the Python PRNG is seeded with the current time and this same seed is used as the directory name for storing test cases in. This ensures that the PRNG seed for each fuzzing session is stored in case the stream of test cases needs to be repeated. By simply hard-coding the value of the PRNG seed to be a seed from a previous fuzzing session, the exact same sequence of HTML test cases will be generated by HTMLfuzz from the grammar.

Inside this directory, the last 20 test cases are stored. Each time a new test case is written to disk, the oldest one is deleted in order to keep only the last 20 test cases and not use up a lot of storage. Since the web browser is not restarted between each test case (this would simply be too slow) it is possible for a crash to be caused by the cumulative effect of a series of test cases. Having the last 20 test cases on disk is therefore useful, and having the PRNG seed also means that we can go even further back in the sequence than this if necessary when investigating a crash.

This handles crash reproduction in terms of the HTML test case. The next part to consider is reproducing what actions the JavaScript took when mutating the DOM for that specific test case.

As mentioned in Section II, the built-in JavaScript PRNG, Math.random(), cannot be seeded by the programmer with a specific value of her own choosing. Many typical DOM mutation actions involve the use of the PRNG, such as selecting random element nodes from the DOM tree. Therefore, even with the HTML test case being saved to disk, without a way to seed the JavaScript PRNG a test case could crash the browser the first time it is loaded, but the crash could not be reproduced as the next time that same test case is loaded the DOM tree would not get mutated in exactly the same way.

A custom PRNG function implemented in JavaScript was therefore needed that would allow for it to be seeded. To solve this problem, the contents of the small xor4096.min.js script was used which is a JavaScript implementation of Richard Brent's xor4096 algorithm taken from David Bau's seedrandom library [33]. This function can be seeded with a custom value that HTMLfuzz stores inside the JavaScript of each test case.

The same seed that is used for the Python PRNG is also used for the JavaScript PRNG. Therefore the seed is kept constant for all test cases across a given fuzzing session, but is different for each session. Each time the HTMLfuzz tool is started, a new seed is generated and then kept constant for all test cases generated in that session. The value of this seed is stored as the name of the directory that the test cases are saved to.

So to summarise, each test case is saved to the filesystem before it is given to the browser, and each test case contains fixed HTML along with a JavaScript containing a fixed PRNG seed. This means that any given test case can be loaded again from disk and should be deterministic, performing the exact same actions each time. There is therefore a good chance of being able to reproduce crashes quite easily without the complexity normally experienced with DOM fuzzing.

When HTMLfuzz is executed by a user, a command line argument is used to specify which TCP port the tool should bind its HTTP server to on the local system. The tool then waits for a web browser to connect to it. Once a web browser connects, the test cases are served continuously until a crash occurs. It is expected that the user will have attached a debugger or other instrumentation to the web browser in order to capture the data necessary for investigating each crash properly.

V. Experiments and Results

Several experiments were carried out to evaluate the new tools that were developed and the methods that they were based on.

Firstly, two corpora of 50 web pages each were created. One was created by random web crawling, the other was generated with the HTMLgen tool. Both corpora were then parsed with HTMLscan. This produced a report for each corpus listing the specific HTML tags and attributes from our specification that could not be found in any of the pages making up that corpus. The resulting numbers from this are shown in Table I.

TABLE I. Evaluating corpora

Corpus Type	Missing Tags	Missing Attributes
Scraped	78	94
Generated	6	0

The shortcomings of building corpora from harvested web pages can be readily seen. Significant numbers of HTML tags and attributes are rare enough to not appear, and this results in a loss of code coverage when fuzzing. The output of HTMLgen is far richer and contains many more different HTML tags and attributes in an attempt to trigger more code paths in web browsers.

The next experiment involved generating a selection of corpora with HTMLgen each of a different size. This started with 10 pages, then 25 pages, followed by increments of 25 up to the largest corpus of 500 pages. Each corpus was then evaluated with HTMLscan to identify at what point all HTML tags and attributes can be expected to be present. This found that by the time the corpus size reached 100 pages every HTML tag and attribute had appeared in it at least once.

The same set of corpora were then processed using a version of the Netsurf web browser that had been compiled using gcov so that it would provide code coverage data. Every time the web browser is shut down it provides a report of the line and function code coverage that occurred during that browsing session. It was found that code coverage flattened out at the point of the corpus containing 200 pages. Generating a bigger corpus than this using HTMLgen appears to be unlikely to give significant coverage gains. Taken with the observations discussed in the previous paragraph, it was concluded that when using the HTMLgen tool, generating a corpus of approximately 200 pages provides the best trade-off between code coverage and speed. Generating a bigger corpus provides little or no coverage increase but will of course slow down the fuzzing campaign as more redundancy is introduced into the corpus and hence into the resulting mutated test cases.

For the next experiment, a corpus consisting of 200 web pages was created using HTMLgen since this had been determined as an optimal size. This corpus was used to seed the Radamsa fuzzer [32] and generate a corpus of 50,000 mutated web pages. This mutated corpus was then processed by a selection of web browsers. This caused six unique crashes to occur, as shown in Table II.

Finally, the HTMLfuzz tool was used to fuzz the same selection of web browsers directly. In this case DOM fuzzing was being applied rather than the file format fuzzing of Radamsa. This experiment also caused six more unique crashes to occur, as summarised in Table III.

TABLE II. CRASHES FROM HTMLGEN WITH RADAMSA

Fuzzing VM OS	Web Browser	Crashes
Windows 10	Microsoft Edge	2
Windows 10	Microsoft Internet Explorer 11	1
Windows 10	Mozilla Firefox	1
Debian 8	Netsurf 3.2	1
Debian 8	Xombrero	1

Out of the 12 unique crashes that were observed during the fuzzing, 3 of them appear to potentially be exploitable. One of these affects Edge, one Firefox and one Midori, all on Windows 10. Further work needs to be done to attempt to develop proof-of-concept exploits from these crashes.

TABLE III. CRASHES FROM HTMLFUZZ

Fuzzing VM OS	Web Browser	Crashes
Windows 10	Microsoft Edge	1
Windows 10	Microsoft Internet Explorer 11	3
Windows 10	Midori	2

The remaining 9 crashes were unfortunately found to be a mixture of NULL pointer dereferences and other issues that are unlikely to be exploitable beyond causing denial of service.

VI. FUTURE WORK

Although the work presented in this paper was overall quite successful, there are a number of ways that the software tools could be improved in future. As we only considered HTML content, and modern web browsers can process many more types of input than this, there is clearly a need to apply the work to other forms of input. Because most popular browsers use third party libraries to handle images, videos, audio files, and also fonts, it is not a good idea to include those in browser fuzzing. Those third party libraries can be fuzzed directly using custom wrappers, which is much faster and more efficient than fuzzing them through the web browser. However, the HTMLgen and HTMLfuzz tools should both generate test cases that include CSS rather than only HTML. We are clearly missing out on code coverage in the rendering engine without this. It would also be worth generating JavaScript within each test case to fuzz each browser's JavaScript interpreter.

Adding CSS and JavaScript would of course involve further work on the Context-Free Grammar after first identifying all possible CSS and JS tokens, in the same way as this was achieved for HTML. It is highly likely that for both CSS and JS, this technique of first identifying all possible tokens including old, deprecated code, and then using a grammar to ensure that all of them are present in the corpus, would once again outperform random web harvesting.

The suite of fuzzing tools could also be improved by the design and development of a test case minimisation tool. Once a test case has been found to cause a crash, it is useful to repeatedly remove parts of the test case as much as is possible without it ceasing to cause the crash. This could be automated after investigating the most efficient way of carrying it out.

VII. CONCLUSIONS

The design and development of the software suite, and then the experiments that were carried out to evaluate it, revealed a number of useful insights into web browser fuzzing. This section summarises the most important knowledge that was gained.

Firstly, a large number of HTML tags and attributes of varying age were identified that are not present in the HTML5 specification but which nonetheless are still processed by many web browsers and therefore need to be included in a corpus to increase code coverage.

It was confirmed empirically that many HTML tags and attributes (both old and modern) are rarely used and hence do not appear often in typical online web pages. This proved that corpora created solely by web harvesting are likely to miss out a lot of valid HTML. It was then proven empirically that there is a correlation between the number of different HTML tags appearing in a corpus and the code coverage obtained from it. It was seen that code coverage rises with the number of different HTML tags and attributes present in the corpus, and then flattens out shortly after the point where the corpus contains each tag at least once. The conclusion can be drawn from this that in order to obtain the best code coverage when fuzzing a browser, and hence increase the probability of finding vulnerabilities, the HTML corpus must contain all existing tags and attributes. Since it was proven that this does not occur with harvested corpora, it can also be concluded that generating the HTML corpus from a well designed grammar is a superior method.

It was seen that seeding the third party fuzzing tool Radamsa [32] with a corpus of 200 pages generated by the HTMLgen tool successfully discovered new bugs in a range of web browsers. Tools such as Radamsa have been used to fuzz web browsers before, as seen in some of the academic literature [22], but had not been used in conjunction with a method of maximising the variety of HTML tags and attributes present in the corpus. This approach was seen to be very promising.

The project also revealed that applying DOM fuzzing techniques to test cases containing the maximum variety of HTML tags and attributes using the HTMLfuzz tool was a successful method for revealing new web browser bugs. Previous work on DOM fuzzing has focused on the manipulation of the DOM tree without putting much emphasis on the quality of the HTML test cases this is being applied to. As with seeding third party mutation fuzzers, this work has shown that it is worth using HTML rich test cases for DOM fuzzing too.

Finally, a promising design for DOM fuzzing tools that allows full proof-of-concept test cases to be saved to disk during fuzzing was presented. This provides a contribution to the field as reproducing and investigating crashes is simpler using the developed HTMLfuzz tool than with previous DOM fuzzers available in the public domain.

Acknowledgment

I would like to thank Professor Howard Chivers of the University of York’s Department of Computer Science for his valuable guidance and feedback during the completion of this work. I would also like to thank Berend-Jan Wever, Mateusz Jurczyk, Atte Kettunen and Rosario Valotta who also provided useful advice and feedback.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[1] Charlie Miller and Zachory Peterson. “Analysis of Mutation and Generation-Based Fuzzing”, 1st March 2007.

[2] Miller, Fredriksen and So. “An Empirical Study of the Reliability of UNIX Tools” in Communications of the ACM, Volume 33 Issue 12, Dec. 1990, pp 32-44.

[3] Rosario Valotta. “Browser Fuzzing in 2014: David vs Goliath”. [Online]. Available: https://www.syscan360.org/slides/2014_EN_BrowserFuzzing_RosarioValotta.pdf [Accessed: 2nd August 2016]

[4] Wade Alcorn, Christian Frichot and Michele Orrù. “The Browser Hacker's Handbook”, Wiley, 2014. Pp 7-8.

[5] Atte Kettunen. “Browser Bug Hunting: Memoirs of a Last Man Standing” at 44CON, 2013.

[6] Jeremy Brown. “Browser Fuzzing with a Twist (and a Shake)” at Zero Nights, 2015.

[7] Rosario Valotta. “Taking Browsers Fuzzing To The Next (DOM) Level” at DeepSec 2012.

[8] Chen Zhang. “Smashing the Browser: From Vulnerability Discovery to Exploit” at HITCON, Taiwan, August 2014.

[9] Renata Hodovan. “Fuzz Testing of Web Browsers” at 3rd User Conference on Advanced Automated Testing, Sophia Antipolis, French Riviera, 20-22 October 2015.

[10] Gadi Evron et al. “Open Source Fuzzing Tools”, Syngress Publishing, 2007. “Chapter 2: Fuzzing – What's That?”, pp 11-26.

[11] Konstantin Serebryany et al, Google Inc. “AddressSanitizer: A Fast Address Sanity Checker” in Proceedings of the 2012 USENIX Annual Technical Conference.

[12] Yan Shoshitaishvili et al, UC Santa Barbara. “(State of) The Art of War: Offensive Techniques in Binary Analysis” in 2016 IEEE Symposium on Security and Privacy.

[13] Charlie Miller. “Fuzzing with Code Coverage by Example” at ToorCon, October 2007.

[14] Michal Zalewski. "Binary fuzzing strategies: what works, what doesn't". [Online]. Available: https://lcamtuf.blogspot.co.uk/2014/08/binary-fuzzing-strategies-what-works.html [Accessed: 2nd August 2016]

[15] Andy Davis. “Fuzzing the easy way, using Zulu”, NCC Group, 2014.

[16] Pedram Amini and Aaron Portnoy. “Sulley: Fuzzing Framework”. [Online]. Available: http://fuzzing.org/wp-content/SulleyManual.pdf [Accessed: 2nd August 2016]

[17] Michal Zalewski. “Mangleme”. [Online]. Available: http://lcamtuf.coredump.cx/soft/mangleme.tgz [Accessed: 2nd August 2016]

[18] John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Pp 163-197.

[19] Tavis Ormandy, Google Inc. “Making Software Dumber” at HIRBSecConf 2009.

[20] Alexandre Rebert et al. “Optimizing Seed Selection for Fuzzing” in Proceedings of the 23rd USENIX conference on Security Symposium, 2014, pp 861-875.

[21] Richard McNally, Ken Yiu, Duncan Grove and Damien Gerhardy. “Fuzzing: The State of the Art”. Australian Government Department of Defence, Defence Science and Technology Organisation, 2012.

[22] Pekka Pietikäinen et al, University of Oulu. “Security Testing of Web Browsers”.

[23] Jared D. DeMott, Richard J. Enbody and William F. Punch. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing" at Defcon 2007.

[24] Fabien Duchene. “How I Evolved your Fuzzer: Techniques for Black-Box Evolutionary Fuzzing” at SEC-T Conference, 2014.

[25] W3C. “HTML5”. [Online] Available: https://www.w3.org/TR/html5/ [Accessed: 5th October 2016]

[26] List of HTML5 Elements and Attributes. [Online] Available: https://www.w3.org/TR/html5/index.html [Accessed: 5th October 2016]

[27] HTML Grammar. [Online] Available: http://docstore.mik.ua/orelly/web/html/appa_02.html [Accessed: 18th October 2016]

[28] Tali Garsiel. “How Browsers Work: Browser Error Tolerance” [Online] Available: http://taligarsiel.com/Projects/howbrowserswork1.htm#Browsers_error_tolerance [Accessed: 26th October 2016]

[29] Griff Ruby. “The Lost Tags of HTML”. [Online] Available: http://www.the-pope.com/lostHTML.htm [Accessed: 26th October 2016]

[30] CodeHelp.co.uk. “Deprecated HTML tags and alternatives”. [Online] Available: https://www.codehelp.co.uk/html/deprecated.html [Accessed: 26th October 2016]

[31] Mozilla Developer Network, “HTML Element Reference”. [Online] Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element [Accessed: 26th October 2016]

[32] Oulu University Secure Programming Group, “Radamsa”. [Online] Available: https://github.com/aoh/radamsa [Accessed: 26th October 2016]

[33] David Bau, “seedrandom.js: seeded random number generator for Javascript”. [Online] Available: https://github.com/davidbau/seedrandom [Accessed: 26th October 2016]

[34] Michal Zalewski, “Investigate cross_fuzz crashes”. [Online] Available: https://bugs.webkit.org/show_bug.cgi?id=42959 [Accessed: 26th October 2016]

[35] John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Page 152.

[36] John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Pp 222-232.