Security Testing of Web Browsers

Pekka Pietikäinen
University of Oulu
P.O. Box 4500, FI-90014 University of Oulu, Finland
pekka.pietikainen@ee.oulu.fi

Aki Helin
University of Oulu
P.O. Box 4500, FI-90014 University of Oulu, Finland
aki.helin@ee.oulu.fi

Rauli Puuperä
University of Oulu
P.O. Box 4500, FI-90014 University of Oulu, Finland
rauli.puuperap@ee.oulu.fi

Jarmo Luomala
University of Oulu
P.O. Box 4500, FI-90014 University of Oulu, Finland
jarmo.luomala@ee.oulu.fi

Atte Kettunen
University of Oulu
P.O. Box 4500, FI-90014 University of Oulu, Finland
atte.kettunen@ee.oulu.fi

Juha Röning
University of Oulu
P.O. Box 4500, FI-90014 University of Oulu, Finland
juha.roning@ee.oulu.fi

Abstract

Web browsers have an enormous install base and vulnerabilities in them can result in wide-spread infections. In this paper we describe efforts made in 2010-2011 to systematically test for vulnerabilities in web browsers. The work was done with Radamsa, a black-box fuzzer that automatically generates test cases based on samples. Approximately 60 bugs were found in widely used browsers, about half of which had potential security impact.

Keywords: Web browser, security testing, vulnerability testing

1 Introduction

Web browsers are the primary means of accessing cloud-based services. Their install base is in the range of hundreds of millions and the market is shared by only a handful of vendors. Ease of injection of malicious data and homogenous environment makes them a good target for malware, while the evolving standards and growing consumer expectations make securing browsers an unprecedented engineering challenge. Browser vulnerabilities have resulted in wide-spread intrusions and the existing exploit kits continue to be a cost-effective means of installing malware.

Vulnerabilities in browsers, like any complex software, are not a new phenomenon. In the past few years, they have been widely categorized as a major risk.
Several factors, such as the huge install base and the move to cloud-based computing, contribute to this. At the technical level, there has been an enormous increase in the functionality required from and provided by browsers and this has also contributed to increasing the number of vulnerabilities. In the beginning browsers were primarily used for displaying hypertext and related images. Current, "Web 2.0", browsers also include or support, e.g., embedded programming languages, such as Java and Javascript, video and plugins. Web 2.0 technologies are also by a number of seemingly unrelated applications, such as map software, iTunes and game consoles, that share the same risks as browsers do.

To provide a seamless user experience, the browsers automatically download and process nearly any data they receive. From a security perspective, this is a nightmare, since this combined with the multitude of data formats supported in modern browsers makes the attack surface enormous. The sheer amount of functionality that needs to be tested makes browsers a complex test subject.

In this paper, we describe work done at the Oulu University Secure Programming Group (OUSPG) in the area of model-inference based software testing, and how this research has successfully been applied to finding previously unknown vulnerabilities in web browsers.

This paper is structured as follows. In the next section, we describe techniques that are used for robustness testing and the issues involved in systematically testing web browsers. Section 3 describes Radamsa, our approach, followed by Section 4, that describes the results of our testing. Finally, conclusions are drawn in Section 5.

2 State of the Art

Software testing is a wide and varied field. Security testing attempts to find faults that could be utilized by external parties to compromise the confidentiality, integrity or availability of the product. As with traditional software testing, security testing tries to uncover faults in the target program, some of which do not have a security impact. Similarly, traditional software testing techniques find faults that do.

Robustness testing is the process of figuring out how well a program can withstand erroneous and potentially malicious conditions. Programs are often fairly well tested against positive requirements, which give requirements like being able to read email messages in Unicode. There are also obvious negative requirements, like not crashing in the middle of writing an email message, and not allowing anyone in the world full control over your account on the computer. Unit testing gives a simple way to approximate meeting the positive requirements, for example by randomly picking a valid state and checking whether it behaves as desired. Negative requirements tend to be harder to check, because they require a program not to be reachable from any of the usually infinitely many starting conditions.

Security testing and software vulnerability detection techniques are traditionally divided into two main categories: white-box testing and black-box testing. Whitebox testing makes extensive use of source code, binary code, runtime tracing or even the modification of the program being tested. The latter one, also called as behavioral testing, can be performed in almost total ignorance of how the test object is constructed. In black-box testing only the interface and the specification of the program are known. The test cases are generated according to this knowledge and the behavior of the program is observed and possible outputs are inspected. There are naturally shades of gray in testing that uses some available knowledge. Intuitively having more data available allows deeper testing of the possible program state space, which while a good thing often seems to lead to testing that sacrifices breadth in favor of depth. [6]

In the security context, white-box techniques include e.g. source code reviews that look especially
for code patterns that are likely to result in security failures, such as the use of easily misused li-
brary calls. Automated static analysis tools, such as Coverity\textsuperscript{1}, can automate this job. The problem,
however, is that they often find thousands of defects in large code bases. A large number of the
reported issues are real bugs, and some of them have security implications. Blindly fixing all of
them is both non-trivial and prone to cause new bugs, as developers may hide the warning instead
of fixing the underlying problem.

One effective way to detect software vulnerabilities is fuzz testing. The technique is based on
generating invalid or random inputs and then injecting them into a program through its interface.
This technique can be used as such with very simple instrumentation to perform black-box test-
ing, usually combined with relatively simple heuristics for detecting obviously erroneous program
states such as memory safety hazards and fatal signals, or it can be paired with more white-box
approaches such as runtime program tracing to gain insight which can be used to control the data
generation.

One of the important properties fuzzing is that one can start testing a program with very little
knowledge about what it does, and gradually refine the testing by giving better samples and adding
better instrumentation to the testing. Another benefit is that unlike in static code analysis, each
issue comes paired with a proof of concept input which can trigger the error, proving that each
issue is accessible from the outside. This is especially important for programs like browsers where
most inputs can be expected to be controlled by malicious parties.

Fuzzing is often divided into two categories: generation-based fuzzing and mutation-based fuzzing.
Generation-based fuzzing \textsuperscript{1} is based on a fixed model of the input space, which is then used as a
basis for building test cases. In some cases, this can be done directly from source code of the soft-
ware to be tested, but often must be done separately. The method works very well with protocols
and file formats that have a formal specification, from which the model can be made from. The
model can also include fields, such as checksums, which are used by implementations to determine
whether the input is valid, and should be processed at all, and without supporting them correctly,
the test case would not have very good coverage. A downside of generation-based fuzzing is that
developing a comprehensive test suite is a major effort, as the test suite is essentially a minimal
implementation of the software to be tested.

Mutation-based fuzzing is, instead, based on samples of data produced by valid implementations.
Simple transformations, such as randomly flipping bits, omitting data or repeating it, or replacing
it with random data, are then performed. It cannot capture all nuances of the data format; this
would require solving many currently unsolved problems in artificial intelligence. Still they are an
important class of fuzzers, because they are exceptionally easy to use and require minimal human
time to operate.

Hybrid approaches also exist. Bekrar et al. proposed a gray-box approach to fuzz testing which is
based on defining vulnerability patterns at assembly level before the actual fuzzing process. The
idea of these patterns is to identify potentially vulnerable code in the binary, for example func-
tions that could lead to memory corruption, and represent them as models. Next, taint analysis is
applied by marking all potentially dangerous data as tainted and tracking their propagation dur-
ing execution. Thus fuzzing can be improved by selecting the most promising test sequences that
are likely to trigger potential vulnerabilities and restrict the test space. The actual fuzzed data is
then generated by applying mutations on sample inputs or by modeling inputs using some learning
methods or the target specification. The execution of the program is observed using systematic
path exploration technique, called concolic or symbolic execution, where the constraints tied to
branch the instructions are extracted and solved in order to explore new paths and discover new
potential vulnerabilities. This strategy has also been used in other fuzzing tools such as Fuzzgrind

\footnote{\url{http://www.coverity.com/}}
and SAGE [8], a white-box file fuzzing tool for Windows applications. In addition, the fuzzing process is iteratively evaluated using code coverage techniques and search algorithms to improve the test case population and expand the coverage. This makes fuzzing more efficient and increases the probability of finding defects and possibly exploitable vulnerabilities. [9]

Security testing of web browsers has been done in the past using all available methods, including white-box tools. The PROTOS HTTP suite [5] tested the robustness of HTTP header parsing of web browsers and servers, and was written in a white-box manner based on the HTTP specification. Due to the wide variety of data formats that browsers have to support nowadays, and the involved code complexity and size, black-box fuzzing is an efficient and simple automated technique for testing browser security and revealing potential vulnerabilities.

One approach to specifically test and improve the robustness of browsers was suggested by Kim et al. They implemented WebDigger, a file fuzzer which utilizes CGI and performs fault injection by creating a HTML document with random data and feeding it to the target when requested. The HTML generator module of the program makes the form of the HTML document. It selects tags and attributes from them and invokes the data generator module to make random data for the value fields of the attributes and contents inside the tags. This way the tags will be untouched by the fuzzer and the fuzzing process does not break the structure of the HTML document. [10]

3 Radamsa

Fully automatic black-box fuzzers are already useful, and the state of the art also has room for improvement. There are commercial products, which produce good testing data for some protocols and file formats, and there are also numerous free tools, which are good at doing some specific thing to some kind of data. We were not aware of truly general purpose data fuzzers, and have repeatedly seen the need for such tools, and thus have been working on one for some years.

Our current approach to automated black-box testing is Radamsa, which is a collection of mutation-based fuzzers that attempt to combine various traditional and novel raw data mutations with some elements of generation-based fuzzers. However, unlike in the traditional generation-based approach, Radamsa uses various automatic techniques to deduce structures from the data, which are collectively called the model of the data, as in generation based testing.

The primary goal is to add some depth to the easy kind of automatic testing in hope that more vendors would start testing their products, and additionally use the tool internally to find bugs in practice, report them and see which tactics seem to work well in practice. The tool combines various tactics in order to maximize the kinds of mutations it can make. The initial version of the tool used several distinct techniques in parallel, which are listed in Table [1], and we are currently researching which would be the cheapest, in terms of time and space, algorithms to find at least the same bugs in order to make the tool more practical.

The fuzzer modules in Radamsa can be divided into three categories. The first category includes only the simplest techniques, namely random generation of data and enumerating. Enumr looks for common short prefixes and after that just starts to enumerate bytes. Range uses the same set of prefixes and generates random data after them.

The second category includes some of the trivial yet very effective techniques include most of the typical sample-based techniques, such as flipr and rambo. They operate on the raw data of the samples without any additional insight into it, and sadly are often effective against poorly tested programs.

The third category includes some novel generation-based techniques such as grammar inference-based fuzzers, grafu and permu, are somewhat more involved. Grafu builds a grammar to model
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grafu</td>
<td>Mutate a grammar inferred from the data.</td>
</tr>
<tr>
<td>fubwt</td>
<td>Mutate the with a disturbed Burrows-Wheeler transformation.</td>
</tr>
<tr>
<td>permu</td>
<td>Permute content around the edges of common substrings.</td>
</tr>
<tr>
<td>rambo</td>
<td>A traditional file fuzzer making simple changes to raw data.</td>
</tr>
<tr>
<td>enumr</td>
<td>Enumerate all byte sequences after a few shared prefixes.</td>
</tr>
<tr>
<td>stutr</td>
<td>Add repetitions to the data.</td>
</tr>
<tr>
<td>tehfu</td>
<td>Build a tree using simple rules and mutate it.</td>
</tr>
<tr>
<td>cutup</td>
<td>Splice and permute the contents of the files.</td>
</tr>
<tr>
<td>flipr</td>
<td>A bit flipper.</td>
</tr>
<tr>
<td>walkr</td>
<td>Make systematically mutations to all positions of all files.</td>
</tr>
<tr>
<td>range</td>
<td>generate chunks of random data</td>
</tr>
<tr>
<td>noise</td>
<td>mix sample data and noise signals together</td>
</tr>
<tr>
<td>forml</td>
<td>generate data from random formal languages</td>
</tr>
<tr>
<td>proby</td>
<td>Fill holes from files by using statistics.</td>
</tr>
<tr>
<td>surfy</td>
<td>Jump between locations with common data.</td>
</tr>
<tr>
<td>small</td>
<td>Test case minimizer. Hopefully not effective if used as a fuzzer.</td>
</tr>
</tbody>
</table>

The data and then proceeds to generate new data from it while randomly adjusting the model and the transcription process. The grammar is built greedily by recursively abstracting out the most shared digram, being a pair of bytes or previously generated rules. The result is a grammar, or a list of lists of binary trees, which efficiently capture much of the redundancy in the data. Being able to make simple mutations on the tree instead of the raw data has repeatedly proved to be a good way to cause changes that would be rather unlikely to pop up from more random raw mutations.

Permu recursively abstracts out a large common substring from the input files and constructs a nondeterministic finite automate (NFA) by merging the strings of the left and right sides of the shared section. For example, given the input strings 'slartibart' and 'bartfast', one will get a language with four words, including the original two, 'bart' and 'slartibartfast'. The shared substrings are search using suffix arrays and multiple sequence alignment.

The different modules tend to find different types of bugs, and their effectiveness varies depending on the format they are used against. For example, text-based formats, such as XML, are often well-structured, and thus grammar inference can detect this structure quite well, but the same applies also for binary data which has enough redundancy in it.

To be effective, mutation-based fuzzers need good samples. This is, however, a balance of effort used for finding or creating the samples and CPU time. To obtain good test coverage, the samples used as a basis should cover the full spectrum of functionality that is provided by the interface that is tested. In black-box testing, this is not always possible, as samples of typical use, which are easily available, may not cover all of the functionality.

Adding more samples improves coverage, but depending on the algorithms used also can slow down test case generation.

4 Results

During the years 2010-11 we have done systematic testing of various open source web browsers, related support libraries and plugins, using our methodology. We started assessing potential for-
mats and protocols, which are summarized in Table 2.

Initially the collection of samples, running them through Radamsa and collecting crashes was quite manual. Even with a limited number of completely randomly selected samples, we were able to quickly find results from nearly all of the tested formats. As the work has progressed, we have continuously automated more of this. In the current iteration, we have a cluster of machines doing testing. New sample sets are simply copied to a central node in different directories, one for each format. The cluster nodes then automatically start fuzzing, and report their findings, including the test case that triggered failures in a centralized location. Some heuristics are done using, e.g., the instruction pointer value, to identify duplicates.

### Table 2. Formats tested and sample origin

<table>
<thead>
<tr>
<th>Format</th>
<th>Description of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>W3C HTML5</td>
</tr>
<tr>
<td>CSS</td>
<td>W3C CSS, HTML5</td>
</tr>
<tr>
<td>PNG</td>
<td>Google and PNG image test suites, manually created and random samples</td>
</tr>
<tr>
<td>JPEG</td>
<td>Google image test suite, random internet samples</td>
</tr>
<tr>
<td>PDF</td>
<td>Vendor web pages, random internet samples</td>
</tr>
<tr>
<td>SVG</td>
<td>W3C test suites</td>
</tr>
<tr>
<td>JavaScript</td>
<td>ECMAScript Test262, SunSpider, JS1k and JS10k demos, test suites (modified)</td>
</tr>
<tr>
<td>GIF</td>
<td>Google image test suite, animated gifs from popular webboard, random internet samples</td>
</tr>
<tr>
<td>Adobe Flash</td>
<td>Random samples from the internet</td>
</tr>
</tbody>
</table>

The samples were initially chosen randomly, but as we continued to test we started utilizing various test suites to gain better coverage. In some cases, we also modified the test cases manually. Modifications were done to increase the impact of the fuzzing to the sample code. Mainly, the modifications were focused on removing comments and minimizing the amount of unnecessary content like page descriptions and advertisements. For some samples it was necessary to add some external files like JavaScript libraries directly into the sample. These suites are described below as examples of material that can be used as samples.

With the current versions of the algorithms used in Radamsa approximately 100-1000 samples that are each under 100 kB are optimal for performance reasons. We have done some initial work using the coverage of samples as a metric for limiting the number of samples by avoiding overlapping samples. For the most part, this has shown to be unnecessary, but may be useful in some embedded environments.

ECMAScript Test262 is a test suite intended to check agreement between JavaScript implementations and the ECMA-262 Standard. "ECMAScript" is the name under which the language more commonly known as "JavaScript" is standardized. The test suite contains thousands of individual tests, each of which tests some specific requirements of the ECMAScript specification. [11]

W3C provides wide variety of test suites for testing the interoperability among CSS implementations. [12] We selected CSS 2.1 and Selectors test suites, because of the state of the test suites and the state of the implementation of CSS specification among the browsers we were testing upon.

The purpose of the W3C HTML5 test suite[2] and Internet Explorer testdrive[3] is to promote interoperability and test the latest features of HTML5, JavaScript and CSS3. Therefore these sets were perfect for our objectives to search for unknown vulnerabilities in the new functionality. Samples picked from 1k and 10k JavaScript demo contests[4] were also quite useful in testing, because of

---

their compact size and creativity.

Also browser benchmarks like Acid3 [13] and SunSpider [14] contained good samples of basic core functions implemented in almost all browsers.

The test cases created by Radamsa from the samples were then run through the browsers. Test cases that caused crashes were collected for further analysis, with duplicates removed automatically. For the most part, post-mortem analysis was done by the vendors. In the past, we have been somewhat sceptical about doing this, as in general, the vendors have had difficulties in taking in bug reports consisting only of the file that caused anomalous behaviour. Browser vendors however have not only been interested in such cases, but have repeatedly promptly fixed them and have often found potential security impact in bugs that were initially thought to be likely harmless. We have repeatedly seen turn-around times of mere hours between filing a bug and the fix landing in the development branch after verification, classification and security assessment.

This kind of ability is rare in the field in general, and requires both automated processes and qualified personnel that are able to process the reports. This is true for both organizations that use mechanisms of this kind in their internal testing, and especially software that is used by the wide public, and that provides automatic crash upload functionality.

Some automated tools for assessing the impact of crashes exist, for example 'exploitable from Microsoft and CrashWrangler from Apple. We have done a similar implementation for Linux as a part of our work. All of these tools are useful in giving an initial estimate for the severity of a given defect, however, the tools are far from fool proof and can result in both serious bugs being downplayed and unexploitable ones being classified as critical.

The total number of previously undiscovered, unique, bugs was 60. Around half of these had a security impact, varying from denial of service conditions in individual browsers to more serious data corruptions in widely used support libraries, such as libpng.

At the end of our test period, the number of crashes in the browsers we are finding has reduced significantly, and are often not reproducible. This alone does not mean that the browsers are robust, but that they are getting more robust against the kind of attacks we have been subjecting them against. We currently have a situation where the programs being tested often misbehave but no longer do so by crashing in an obvious way. Initial experiences testing have shown that Radamsa still is able to identify new bugs, but the test subjects have to be instrumented better, which usually requires recompiling the code with debug options that have a significant performance impact. Compiler extensions, such as Google Address Sanitizer [15] appear to be state-of-the-art in this area. This might be a result of the so-called pesticide effect as described by Beizer [6]. Certain type of testing makes the test targets more immune against that particular type of testing in the long run. To obtain more bugs perhaps a different set of samples or techniques should be used in the same targets.

5 Conclusions and Future Work

Automated model-inference based fuzzing is an effective way of finding vulnerabilities in large, complex, code bases that contain many interfaces, such as web browsers. Our approach, Radamsa, has a proven track record in this area.

In the future, we will try new areas, such as web-based cloud services, as test subjects. Sample selection based on coverage has also shown to be a promising area for further work, but with many products, results can be found using even a few, completely randomly chosen, samples.

Algorithmic improvements to the fuzzers and the addition of instrumentation to more effectively
capture misbehaviour of test subject may allow Radamsa to be more effective at hitting bugs, but the biggest hurdle is getting organizations to start any kind of fuzz testing. We have tried to make the hurdle as low as possible, and aim to make it even lower in the future. Ideally, all software would be continuously tested, automatically, in the cloud.

References