Fuzzing JavaScript Engines with Semantic Aware Tools

CSCI 5271 Fall 2019

Lee Wintervold
University of Minnesota
winte413@umn.edu

Sam Hanson
University of Minnesota
hans5376@umn.edu

Daniel Rockcastle
University of Minnesota
rockc004@umn.edu

Daniel Belair
University of Minnesota
bela0019@umn.edu

Zaffer Hussein
University of Minnesota
husse147@umn.edu

ABSTRACT

JavaScript (JS) is one of the most widely used languages on the web today. While most prominently recognized in its ability to create dynamic websites, it can also be used to build more complex systems. Browser engines are the programs that allow for JS execution and are built into popular web browers such as Google Chrome, Safari, Firefox, and Internet Explorer. Due to their complication and large codebase, they can have serious security flaws which in some situations allow for remote code execution. In order to find some of these interesting crash states we used CodeAlchemist which is a fuzzing tool for JS engines. This tool creates semantically-correct JS files that, when executed by an engine, may cause it to crash. Due to the nature of JS's implementation and its popularity, JS engines provide an interesting opportunity for further exploration. This paper presents how we implemented and extended a fuzzing tool and our findings which include JS files that cause JavaScriptCore, the engine behind Safari, to crash when executed.

KEYWORDS

JavaScript, Fuzzing, Code Binding, Static Checkers, Vulnerabilities

1 INTRODUCTION

JavaScript is a tool used in ninety-five percent of today's websites [11]. It is also the number one language on GitHub [8]. Due to core features and design choices, JS is a point of introduction for numerous vulnerabilities and crash risks in applications. The popularity of JS is a result of its versatility with many applications ranging from server-side scripting to the creation of dynamic and interactive websites. Its omnipresence and popularity make it a likely target for attackers.

JavaScript code can be run in complex environments where it interacts with components written in different programming languages. Additionally, web browsers and their underlying JS engines, extensions, and components exhibit a great deal of variation and possible combinations. For a given web page or application, behavior of one browser may be entirely different from that of another browser. These differences in behavior and outcome can be unpredictable; they can only determined through thorough testing or observation. All of these factors combined create a large domain to explore. This is true for both those seeking to create, maintain, and present information on these systems and also for those who seek to exploit or debase them.

In our research, we initially examined the idea of finding bugs in JavaScript binding code through static analysis. As our research progressed and we discovered which tools were and were not available, our work evolved into examining how we might fuzz JS engines. While our final results are not specifically related to bugs in JS bindings, we feel that it was an important part of our journey to understand the language, tools, and methodology. Based on what we learned from the resources we used in this project, we observed a pattern indicating that the discovery of advancements in the development of these tools is accelerating. As research progresses, work is being done on improving the methods to locate errors in any language, not only JS, by creating more lean and efficient tools. Our final project involves using and extending a fuzzing tool, CodeAlchemist, which seeks to expose vulnerabilities in JS with a novel approach to fuzzing.

As we discovered new and related topics in cross-referenced papers, we were able to explore a wider range of JavaScript vulnerabilities and issues. We learned about different methods that researchers are using to discover weaknesses in JS and other languages. All of these advances are coming together to create safer, more efficient, and more concise websites and applications. Static checking tools required thousands of lines of code. Through the development and refinement of micro-grammars, the amount of code to create a working product has decreased by orders of magnitude [1]. Fuzzing applications used in the past often had large amounts of data and computations that were not always used or needed. These programs were often employed blindly or "dumb" in reference to what was being fuzzed. The limitation of fuzzing to quality seeds that were more likely produce results is leading to leaner seeds with a better rate of success and fewer false positives.

JavaScript limitations and problems

The dynamic nature of JavaScript as a programming language can be the source of many bugs. At runtime, JS programs can modify the type system and dynamically generate code. This dynamic nature can make static analysis difficult. Variables may be assigned at least one of seven ECMAScript standard JS types at runtime: *Boolean, Object, Null, Number, Undefined, String, and Symbol.* Specific JS engine implementations may define more built-in types. JS iteration and use of ”for each” is treated differently and can raise inconsistencies when bound to C++ iterative for loops, especially when also considering the memory implications that may arise in these cases. Runtime errors commonly occur during evaluation of JS statements because of its dynamic typing system. One limitation of static checkers is the inability to examine what occurs at runtime. Because of this, a different type of examination tool such as a fuzzer must be used to catch runtime errors.

Binding code

Often, the functionality of JavaScript is extended in the browser through underlying engines. These engines vary from browser to browser and are written in lower level languages such as C++. JS is able to do this through a binding layer in which JS functions call C++ functions with JavaScript arguments. Although JS does not suffer from the lower-level types of vulnerabilities present in C++, these vulnerabilities exist through the binding layer functions. Typing in C++ must be handled by the programmer and explicitly stated in the code. Memory management is deliberately handled by the programmer in C++. Memory must be allocated explicitly within programs. When memory space or variables are no longer used, it must be freed by the programmer. Access of memory that has been previously or prematurely freed is possible if the program does not have safety measures in place. These types of errors can cause null pointers or use-after-free issues and lead to program crashes or memory exposure which an adversary can exploit [1]. While C and C++ can be faster, more secure, and more powerful languages, it is the responsibility of the programmer to write code in a correct and secure manner. Furthermore, the introduction of binding code may introduce new potential vulnerabilities. It is possible to produce valid JS code that in itself is innocuous, but presents vulnerabilities or errors when introduced to a binding layer. The original code could be secure within itself and the authors may still have no way to know if the introduction of binding at a later time will create a new vulnerability.

Finding the vulnerabilities

Static checkers do not have the ability to fully examine code and find all of the bugs that are present at execution time. Because of this, other tools such as fuzzers are sometimes a better option to find errors and potential issues [1]. Static checkers analyze binding layer code to catch vulnerabilities which can be separated into three categories: crash-safety, type-safety, and memory-safety bugs [5]. For crash-safety violations, the checker flags all hard-crashing asserts which rely on user supplied JS arguments [2]. This is done by analyzing each binding-layer function to check if a JS variable is used in a call to a hard-crashing macro such as `ASSERT`. The type-safety checker determines if JS arguments are correctly converted to the corresponding binding function types. This is done again by analyzing each function and hosting a set of the JS values that have not been type checked. When a value is cast to the binding layer type without being type checked, meaning it is still in the set, a flag is raised. Memory-safety violations are caught by a similar process to the type-safety checker. Values that have not been type checked are added to the set, and if they are used in memory operations a flag is raised. This approach is very efficient, but the flags raised are not always exploitable. In addition, creating a static checker is very involved (often hundreds or thousands of lines of code) and can still miss a lot of bugs. The results found with static checkers can vary heavily because of the patterns and grammar used to implement them.

2 ERRORS IN JAVASCRIPT BINDINGS

When a bound JavaScript method is called, the corresponding C++ binding layer function is executed with the arguments supplied to the original JS function. As described above, this translation can go wrong. An example of this is when C++ code does not check the size of the buffer passed in from JavaScript which can lead to a buffer-overflow attack. Binding layer functions are also programmed to hard-crash in certain cases. This can lead to denial of service attacks which are detrimental to the host. Developing binding code is already hard enough [2], and these issues alongside memory vulnerabilities can be difficult for developers to avoid. Even if a JS programmer writes clean code, there can still be vulnerabilities because their JS code interacts with vulnerable binding code. The perfectly legitimate JS code may have a memory segment that is no longer used and garbage collected. In the bound C++ base code the corresponding memory may continue to exist. It is very difficult to catch these vulnerabilities at the JS language level.

3 A CHANGE IN DIRECTION

Once realizing we could not get the source code to the static checker, we decided to move in a different direction. Although we did not look specifically at JavaScript binding issues or use static checkers in our final implementation, the study of each of these areas moved us forward in our understanding. We gained insight into how to "think like an adversary" through the examination of these subjects. Understanding the nature of static checking tools complements our understanding of the fuzzing tool we worked with. Static checkers may not be able to easily find typing problems that may occur at runtime with JS code and bindings. By contrast, a fuzzing tool is more dynamic and can perhaps more easily find these types of problems. Learning about both approaches and implementations widened our understanding and gave us a better idea of what to look for while examining programs and code.

4 INTRODUCTION TO FUZZERS

One alternative to static code analysis is fuzzing. Fuzzing uses a black box method to test software against a number of random inputs. Generally, large numbers of random inputs are provided to software and the resulting actions are observed and recorded. Further testing is done with input to the test program that is similar to what caused unexpected results by the fuzzer. Using an iterative process with this method may reveal more information about the software being tested, its behavior, and potential vulnerabilities. Refinements can be made to optimize fuzzers by using input that is not simply random. Many cases can be excluded as obviously syntactically incorrect within a language or program that is being tested. Developing smarter fuzzing software has been found to increase the effectiveness of these programs. These fuzzers forego the use of incorrect or broken statements and which will generate no viable results. While more effective, the development of smarter fuzzers can take a significant amount of time and lines of code for the developers to create. The manual effort of these can be great and must be repeated for different sets of applications and problems.

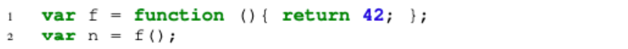
In the case of JS engines, generation-based fuzzing involves taking a file and breaking it into code segments to use as combinable test cases with random inputs. The current state of the art fuzzers are Langfuzz and jsfunfuzz. The two combined have found more than 5,000 bugs in JavaScript engines since 2006 [4]. Jsfunfuzz works in a similar fashion to other JS fuzzers, but it does not use a seed file. It instead generates random JS code snippets to use as test cases against the engines. jsfunfuzz has been very successful in its bug-hunting career through the use of syntactically valid generated JS code snippets. As mentioned in their paper [4], the issue with the modern engine fuzzers is that their generated test cases often raise runtime errors because they fail to construct semantically valid blocks. Constructed code blocks may have type issues or use of undefined variables. CodeAlchemist takes this into account by breaking JS files into code bricks and analyzing the dependencies of each brick, essentially creating a rule-set for which bricks can be combined to create semantically and syntactically valid JS code to fuzz. Our approach utilized CodeAlchemist as a fuzzer to streamline the bug finding process. Static analysis tools are often built to discover particular types of vulnerabilities, and are not designed to report about any other types. In contrast, fuzzing tools are less targeted and are able to find vulnerabilities of many different types. This can be due to several reasons including binding layer code that causes unexpected side effects or, in our case, type confusion when the JS code is executed via the engine. It is difficult to differentiate which bugs can be triggered with JS code and to craft the necessary code to trigger the bug. Our approach was to measure the effectiveness of the CodeAlchemist fuzzer by testing against older versions of popular JS engines such as V8, ChakraCore, and JavaScriptCore to see if the known bugs for those versions can be found and if any new bugs could be discovered.

Methodology

The crux of CodeAlchemist's approach to fuzzing is its ability to assemble semantically correct JavaScript code to fuzz against the engine. The end results are test cases that are mostly free of runtime errors that plague other fuzzers. The approach involves splitting a seed into code bricks which are then labeled with constraints. These constraints are guidelines for combining bricks that will avoid runtime errors. Code bricks are given two constraints: preconditions and postconditions. Preconditions state which symbols need to be defined prior to avoid a runtime error. Postconditions state which symbols are guaranteed to be defined at the end of the code brick. If the postcondition and precondition of two code bricks match, then their combination produces a semantically valid code snippet. Figure 1 shows an example of two compatible code bricks. This snippet is in itself a code brick, and can be further combined with other compatible bricks. These combinations are what is run through the fuzzing operation. Semantically incorrect seed files are discarded during the seed preprocessing step. Bad seeds such as this are unlikely to produce any reliable result. These are a waste of memory and computation. To that end, the ability to refine a fuzzing tool and its output saves computation time and ensures a greater probability of higher quality results.

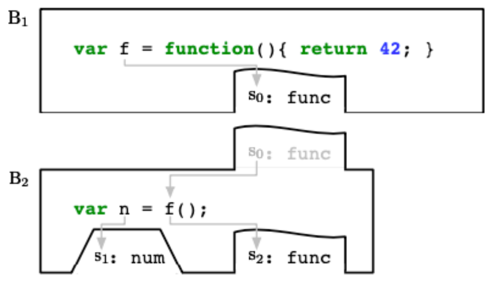
5 CODEALCHEMIST ARCHITECTURE

Figure 1: Code Bricks



```
var f = function (){ return 42; };
var n = f();
```

(a) A sample JS seed with two statements. We assume that we create one code brick for each statement, respectively.



(b) Two code bricks obtained from the seed. The teeth and holes on each code brick represent the assembly constraints.

[4]

CodeAlchemist takes an input of seeds and a JS engine or program to fuzz against. The seeds are formulated into a parsable Abstract Syntax Tree *AST* with Esprima. The tree models the symbols and flow of the code and creates syntactically valid JS code fragments. There are multiple ways to split the AST into code fragments. The CodeAlchemist authors had to decide which would be most beneficial given their approach to fuzzing. One approach is demonstrated in the Langfuzz fuzzer in which they break the AST into subtrees. An inefficiency with this approach is that the code bricks would repeat as they are constructed from non-terminal nodes (meaning non-leaf nodes). You would often find that a code brick may be the subtree of another code brick, which means they share a lot of the same code. The CodeAlchemist authors decided to take a different approach to avoid a pool of similar or redundant code bricks. They decided to break code bricks into individual JS expressions since this is the smallest fragmentation that results in a syntactically valid statement. This approach for fragmenting the AST is implemented in the Seed Parser module. The Seed Parser parses the AST representation of the seed file to construct a set of code bricks. During this process, code bricks that are detrimental to the experiment are removed. For example, some bricks may contain crash macros that would terminate the program. Other examples include the `eval` functions and `no-ops`. They also take the trouble to remove any bricks that are syntactically incorrect by running them on the target JS engine to see if any syntax errors are caught.

Figure 3 displays how an ordinary JS code snippet (Figure 2) may be represented within an AST [9] The constraint

Figure 2: Sample JavaScript function

```
function foo(x) {
    if (x > 10) {
        var a = 2;
        return a * x;
    }
    return x + 10;
}
```

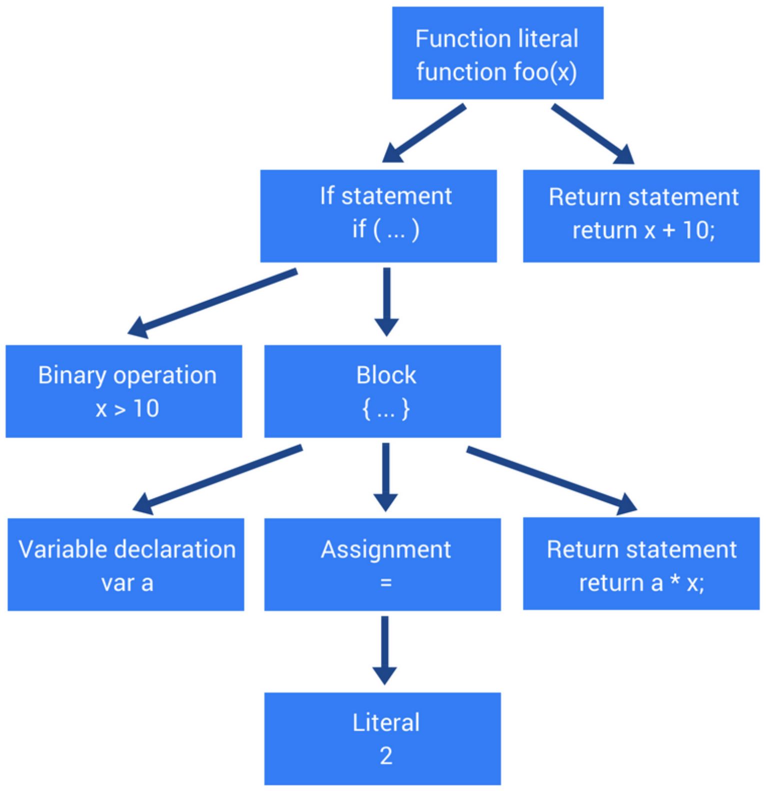
analyzer is then given these code bricks which determines what is defined and used within each brick to create the constraint tags for the code brick. This step is crucial in the CodeAlchemist architecture because it ensures that semantically correct test cases are being assembled. The analyzer performs a static analysis of each code brick and also defines the types of each variable at the end of each code brick. CodeAlchemist must determine all the builtin types of the target it is fuzzing against, which extend beyond the seven primitive types of JS defined in the ECMAScript 6 standard. In addition, the typing system of each engine may have subtle differences. For instance, variables in separate code bricks both may be of type `object`. This does not mean they are the same object or object instance. If CodeAlchemist were to infer that, it would lead to runtime errors as object variables defined in one code brick may be used with a different object in another code brick. CodeAlchemist handles this by logging the subtypes of the 'object' type. In order to actually know what type variables in a code brick are, each brick is type-logged at the beginning and end. The logging function takes in the list of variables in the brick and returns a mapping of their types. There are cases where the type of a variable is dependent on the result of an `if` statement, and in this case these variables are given a union type of their possible types. This information is then used to create the preconditions and postconditions of every code brick. These tagged code bricks are then added to a pool for fuzzing.

Figure 4: Example code brick assembled by CodeAlchemist

```
var firstName =     John
var fullName = firstName + lastName
```

The pools are passed along to the engine fuzzer which is responsible for generating the test cases for fuzzing. The initial step of the engine fuzzer is to combine code bricks given their assembly constraints. In the Figure 4 segment above, we see a brick ready for testing. This code brick cannot be run in

Figure 3: Abstract Syntax Tree.



isolation because the variable *lastName* has not been defined. In traditional JS fuzzers, the majority of the issues raised from the fuzzer are runtime errors. The assembly constraints for this code brick would prevent such a thing by ensuring the variable *lastName* is a postcondition to the brick prior to it. It also needs to ensure that *lastName* is of type string or else the '+' operation would result in a runtime error. This is also where CodeAlchemist's configuration parameters (i_{max},p_{blk},i_{blk},d_{max}) come into play. We did not specify these so the default values were used. They determine how blocks are assembled and combined. The execute function then fuzzes the generated test cases on a JS engine. The algorithm starts with an empty code brick B and the code brick pool. It keeps appending code bricks to B until it reaches i_{max}, one of the user-configurable parameters. Each time a code brick is appended to B, it is treated as new a test case. At each iteration, the algorithm also decides whether to append the regular code brick C that was chosen or to reinvent C and append. This is s probability based decision given by p_{blk} which is another user-defined configuration parameter. It chooses to reinvent the statement with probability p_{blk}, or use the regular statement with probability 1-p_{blk}. *iBlk* defines the maximum number of iterations for generating a block statement, and *dMax* defines the maximum nesting level for a reassembling block statement.

The code snippets that cause the engine to crash are saved by CodeAlchemist and provided in a log file. The programmer can then use this information to further analyze the cause and nature of the crash within the engine. The fuzzer will not find every bug, but the chances of finding a bug increases with compute time and seed choice. The quality of seed selected is a larger factor in the success than the quantity of seeds selected and employed [10]. We found the topic of seed selection to be a vast topic that could have its own report. Within the domain of our resources, we were limited with the seeds we had available and could employ. Due to the nature of CodeAlchemist's approach to fuzzing, we feel that it would be interesting to test, measure, and report the differences in results found with different sets of seeds. Breaking the seed sets down further to sub-sets or sub-groups and examining the results could be an additional area of study.

6 METHOD

Software and hardware limitations as well as availability drove our direction.

Software

We contacted the creators of the static checking tool Microchex (*µchex*). *µchex* implements the idea of "micro-grammars" in order to create more effective checking tools with fewer lines of code [2]. After contacting the authors who were most involved with the creation of *µchex*, we found that it was not available. They did note that an open source version of the software would be released within a year. In addition, they provided us directions to a GitHub repository with some exploits as potential seeds. We were also unable to locate a viable static checking alternative and the implementation of our own static checker would be outside of the scope of this course.

CodeAlchemist was discovered in researching one of the referencing papers. A GitHub repository showed that the source code was available. We contacted the authors of the CodeAlchemist paper and they offered some initial insights including suggestions for seeds. CodeAlchemist was created with source code written in C and also in the functional programming language F. An advantage to our approach for this project was the fact that the implementation of this program would require little to no alteration of the source code. The nature of this program meant that we could extend the original paper's ideas through altering seeds, targets, and run parameters and variables.

Hardware

CodeAlchemist was implemented by the original authors on a machine equipped with 88 cores and 512GB of main memory [4]. It must also run on a computer with Ubuntu 18.04 (or newer) Linux. We had originally installed CodeAlchemist on personal computers with VirtualBox software running an Ubuntu Linux environment. This created the capability to create an image of a virtual machine with the software. It also offered some portability to the program, allowing us to set up the image on our local machines without knowing exactly what machine we would eventually run on. The major limitation was that our personal computers were limited to four cores in most cases. A number of options were explored, including free tier cloud computing accounts (AWS/Google Cloud) which would run our Ubuntu image. We found a better solution at the Minnesota Supercomputer Institute (MSI) located in Walter Library on campus. Once the use of the MSI cluster was authorized, we were able to find online resources that guided us on how to use the services. In addition, members of the team attended an informational session put on by MSI at their facility in Walter Library. The session was open for different users and one-on-one questions. We found the information provided to be especially helpful and the ability to ask direct questions to the staff at MSI moved our work forward a great deal.

Our experiments

CodeAlchemist was run on the MSI Mesabi small queue, containing 216 total CPU cores of Haswell E5-2680v3 processors. Jobs were run in 24 hour batches over a three week period. Since Mesabi runs CentOS 7 and CodeAlchemist requires Ubuntu, Singularity containers with Ubuntu 18.04 installed were used to create the required environment. The current engines fuzzed against were ChakraCore (v1.11.15), V8 (commit d8ba2856f4c9ced82d652876c963d5f03094af31), JavaScriptCore (commit 09866a9ce8eda88936bfc27325 21547 e49af32ab) and Node (v14.0.0). The old engines fuzzed against were ChakraCore (v1.10.1.0), V8 (v6.7.288.46) and JavaScriptCore (v2.20.3). The current browser engines were fuzzed with a single seed set built from combining all exploits in js-vuln-db [3]. Node was fuzzed with seeds from the parallel test suite in the NodeJS repository, along with js-vuln-db V8 exploits. For the older engine replication runs, each engine was fuzzed with JS files from the test suites in their respective repositories.

ChakraCore, V8, and JavaScriptCore were fuzzed for two weeks. Node, ChakraCore (2018), and V8 (2018) were fuzzed for one week; JavaScriptCore (2018) was fuzzed for two days. Due to time constraints and issues with interfacing with CodeAlchemist, the SpiderMonkey (Firefox) engine was not used in our experiments.

7 WHAT WE FUZZED AND WHY

The original selection of experiments included fuzzing current and older versions of these engines with seeds from the respective engine test suites. We estimated that this method would not likely report new findings, because this was the seed selection of the original CodeAlchemist authors. Given the complexity of the of the CodeAlchemist program and the novelty of using the computing array, care was taken to ensure that we could get the program to run correctly and that we could get results of any type.

Js-vuln-db

The choice of js-vuln-db [3] as a seed set for running against current engines was motivated by the "security quality" of the seeds. While seeds drawn from a full engine testing suite are likely to have reasonable coverage of the associated engine, in general most of these seeds are not interesting from a security standpoint. The nature of CodeAlchemist's exploit generation is such that it only makes do with what it is given; CodeAlchemist in a certain sense shuffles and recombines JS snippets from the seed set, and will not generate novel JS.

So starting from a seed set which intentionally is trying to break aspects of the engine is an especially good choice for a fuzzer like CodeAlchemist.

Node extension

The codebase of CodeAlchemist was extended to fuzz against Node. This was done by directly modifying the codebase of the fuzzer in a fork of the CodeAlchemist repository. Overall the changes were not large, although the process took some time. The largest hurdle was overcoming failures in the rewriting and instrumentation steps of seeds in the CodeAlchemist workflow, as they require the names of builtin types for the target engine/binary. For the regular engines, running a small JS script against the engine is sufficient to extract these types. Node would produce the correct output when the script was run in a REPL, but would give any output when given the script as a command line argument. To resolve this, the Node types were manually extracted and hardcoded into the source code. Finally, CodeAlchemist uses the Esprima parsing tool to parse all JS seed files, which expects all syntax to be ECMAScript 6 compliant. Not all Node syntax is ES6 compliant, and our extension did not address this issue, so not all Node seed files could be utilized for the fuzzing that was performed.

8 RESULTS AND ANALYSIS

The results of our experiments are summarized in Table 1 and Table 2.

Runtime errors

Reported segfaults of this type include reference errors, type errors, syntax errors, and "out of stack space" errors. In general all the errors associate with running files in this category are issues with CodeAlchemist producing invalid JavaScript rather than a bug in the target. We suspect that there are improvements to be made to the parsing step of CodeAlchemist that would address the generation of these files, since the original authors outsourced the actual parsing to Esprima and focused on the semantics-aware construction.

For Node, although we did not implement correct parsing for Node's syntax extensions to JavaScript (e.g. top-level returns), the runtime errors encountered are likely not due to bad seeds since the total number reported is reasonably close to the original engines, for which the syntax parsing accepted all input. Node seed files that Esprima failed to parse are simply discarded by CodeAlchemist in the seed processing step, so extending the acceptable syntax would simply allow for those rejected seeds to be included.

Could not reproduce

Reported exploits of this type are likely intermittent failures. If an exploit has a low probability of triggering, potentially due to race conditions, a segfault would not always been observed. These failures were ran only a handful of times, so a bug with a low probability of triggering could have slipped by undetected.

Timeout

The configuration options for CodeAlchemist include a parameter which defines the maximum amount of time a particular potential exploit can be run. The default value of thirty seconds was used for all runs. Any attempted crash CodeAlchemist runs that executes longer than this is reported as a bug. It is unlikely that any of these files actually contain a vulnerability. Ideally, all generated JS files would be run to completion in order to determine if they trigger a bug, but keeping the timeout reasonably low is important to ensure that CodeAlchemist does not expend a large amount of time on files that are not crashing. We believe that a low timeout period will lead to finding more vulnerabilities. The reasoning is that the more serious issues are discovered quickly. Discarding files that have been running for longer than thirty seconds in order to be able to explore more items improves the probability of finding a high number of vulnerabilities. Long run files still present a possibility of discovery, but the probability is likely lower than that of finding some in the remainder of the field.

Segfaults

Segfaults were found for ChakraCore, JavaScriptCore, and ChakraCore (2018). Running each JavaScriptCore crash file in a debug build resulted in finding five "types" of failures. Five specific lines in the source code correspond to where the crash occurred. The idea was to see how many different bugs CodeAlchemist actually found since there are several different ways to trigger these bugs. The remaining fifty-nine segfaults in the release engine tested did not segfault in the debug build. Our analysis focused on Type A crashes, as they counted for the largest number of exploits found during fuzzing.

JavaScriptCore background and optimization

In order to increase the execution speed at which JavaScript is ran, the JavaScriptCore(JSC) engine implements 4 execution tiers. These tiers differ in execution and optimization strategy. The lowest level of optimization is called the LLInt interpreter, a simple JS interpreter. A more aggressive strategy, Just-In-Time (JIT) compiling, is introduced to the following optimization levels. JIT compiling turns JS into machine code, significantly increasing execution. The JIT layers are introduced to JavaScript code which is "hot" or called/executed often. Baseline JIT is used if the code executes 100x or more in a single run, Data Flow Graph (DFG) JIT is used if code executes 1000x, and Faster Than Light (FTL)

Table 1: CodeAlchemist reported crash files.

Engine	Runtime Error	Could Not Reproduce	Timeout	Segfault
ChakraCore	9	9	4	1
V8	2	1	4	0
JavaScriptCore	9	13	2	137
Node.js	8	3	3	0
ChakraCore (2018)	0	1	5	4
V8 (2018)	16	6	2	0
JavaScriptCore (2018)	0	0	0	0

Table 2: JavaScriptCore debug build segfault breakdown.

Crash line	Count	Class
Type A	52	IndexingHeader
Type B	17	StringImpl
Type C	7	Structure
Type D	1	ThrowScope
Type E	1	MarkedBlock
No crash	59	

JIT is the most aggressive optimization level, using C-like strategies. However, these strategies do not come without tradeoffs. For instance, with increasing aggressiveness comes decreased security and stability. While turning JS code into machine code does greatly increase speed, it also removes type-safety checks that the interpreter level contained. This means type confusion vulnerabilities become a major concern as JavaScript is a dynamically typed language. If an adversary was able to confuse the engine, arbitrary code execution is possible under the right conditions. While we were unable to find a vulnerability allowing code execution, one of the files we analyzed did show signs of a possible arbitrary read, as described in LiveOverflow's tutorial [6].

JavaScriptCore crash files

Due to the numerous JS files CodeAlchemist produced when run against the JavaScriptCore (JSC) engine, we chose to analyze each one by classifying them into groups. All the segmentation faults that were reproducible in the debug build are listed in 2. We will specifically focus on Type A crash files. The segmentation fault was thrown on line 56 in a function that returns the vector length of a butterfly. A butterfly is an internal JavaScriptCore data type that allows for storing properties and elements of an array in the same region of memory. It is clear that a pointer to a butterfly is getting corrupted and an invalid memory address is trying to be accessed. The difficult part is tracing the root cause of this crash and is out of the scope of this paper. Further investigation and research is needed as that will determine if an exploit is possible through this particular bug.

Crash types B through E were not investigated further due to time constraints. These crash types represented a smaller proportion of total crashes compared to Type A.

The remaining 59 segfaults in the release build did not crash in the debug build. Each of the crash files either ran into a runtime error, timed out, or simply completed execution. These crashes were not looked into further because investigating these crashes with only the release build would be prohibitively difficult.

ChakraCore crash files

The crash we found is due to a `SIGILL` instruction, similar to CVE-2019-0609 [7], where a function is defined, gets overwritten, and then executed. In our crash, the `SIGILL` is preceded by an OutOfMemory error. However, a `SIGILL` failure is indicative of attempting to execute a region of memory which does not actually contain program instructions. If this region of memory can be controlled, it should be possible to execute arbitrary code. Unfortunately, the crash we found does not fail in the same fashion in the debug build of ChakraCore, so additional effort will be required to decipher the exact chain of events in the release build.

V8 and Node extension

Our choice of seeds and fuzzing technique were not able to extract any segfaults in either V8 or Node.js. Since Node is built over the V8 engine, and we ran Node with V8 js-vuln-db seeds, it is expected that any bugs would be likely observed in both. However, none were found in either target, which mostly indicates that V8 is more hardened against attacks similar to previous vulnerabilities from js-vuln-db. The failure to find segfaults in Node can be attributed to both the relative hardness of V8, and issues with Node seeds. The issues were twofold: Esprima's syntax does not include all valid Node syntax, so many test files were discarded in the preprocessing phase, and not all tests from the associated repository could be used in isolation due to requiring includes, requiring certain directory structure, and configuration files outside a particular JS test determining how such a test will be run. Although our experiments were not able to find segfaults, these issues would need to be addressed before concluding that CodeAlchemist is not able to find vulnerabilities in V8 and Node with the used seeds.

2018 engines

The replication portion of our experiments were not successful. Although four identical segfaults were found in ChakraCore (2018), they did not correspond to any of the exploits found in the original CodeAlchemist paper. It is likely that the most significant reason for this failure is related to extracting seeds from corresponding engine repositories. As mentioned for the Node.js extension, pulling files out of testing directories is not as straightforward as simply copying all the JS files; there is information stored in the directory structure, includes, and config files which must be accounted for to totally utilize a testing suite. The seed sets used for the 2018 engines were subsets of the testing directories which could be run "straight" on the corresponding engines; these JS files did not require any of the structure of the testing directory or surroundings. The results found are consistent with our seeds having less coverage than the original authors' work.

9 CONCLUSION

CodeAlchemist was successfully extended to fuzz against Node.js, although there is remaining work to allow for seed files which include Node's extensions to the syntax of the language. We believe that Node may provide the opportunity for more interesting findings due to the nature of it's implementation compared to the other engines mentioned. The full results of the original CodeAlchemist paper were not replicated, likely due to seed limitations. SpiderMonkey was also not fuzzed against due to time constraints.

We also managed to run CodeAlchemist successfully against a novel seed set, and produced a number of crash files which we are still investigating. Further research is needed to determine if any crash files could be used as a base for an exploit.

We found semantic-aware fuzzing to be an effective approach, although it seems to be highly dependent on initial seeds. Continued utilization of new CVE seeds would be ideal to continue the work of this paper.

Responsible Disclosure

We intend to disclose the vulnerabilities we found in the current engines to the appropriate entities (Microsoft for ChakraCore, Apple for JavaScriptCore). There is still work to be done to determine the root causes of the crash files we found, and we intend to spend more time on this prior to disclosing our findings. Ideally root causes would be found for every reproducible segfault in the debug build, and we commit to sending our findings before the start of the 2020 Spring semester (Jan 20th).

REFERENCES

[1] Fraser Brown. 2016. Short paper: Superhacks: Exploring and preventing vulnerabilities in browser binding code. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 103–109.

[2] Fraser Brown, Andres Nötzli, and Dawson Engler. 2016. How to build static checking systems using orders of magnitude less code. *ACM SIGPLAN Notices* 51, 4 (2016), 143–157.

[3] Choongwoo Han. [n. d.]. js-vuln-db. https://github.com/tunz/js-vuln-db

[4] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.

[5] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. 2015. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[6] LiveOverflow. [n. d.]. Browser Exploitation. https://liveoverflow.com/tag/browser-exploitation/t

[7] mitre.com. [n. d.]. Common Vulnerability Exposures. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0609

[8] multiple. [n. d.]. The State of the Octoverse. https://octoverse.github.com/

[9] Lachezar Nickolov. [n. d.]. How JavaScript works: Parsing, Abstract Syntax Trees (ASTs) + 5 tips on how to minimize parse time. https://blog.sessionstack.com/how-javascript-works-parsing-abstract-syntax-trees...-abfcf7e8a0c8

[10] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[11] w3techs. [n. d.]. Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript