BINARY BROTHERHOOD

WE SAY WHAT WE DO, AND DO WHAT WE SAY

WHITE PAPER
Introduction to Fuzzing JavaScriptCore on MacOS with AFL++

Version 1.1

BINARY BROTHERHOOD, INC.

binarybrotherhood.io

TABLE OF CONTENTS

INTRODUCTION

"Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. Typically, fuzzers are used to test programs that take structured inputs. This structure is specified, e.g., in a file format or protocol, and distinguishes valid from invalid input. An effective fuzzer generates semi-valid inputs that are "valid enough." The parser does not directly reject them but does create unexpected behaviors deeper in the program and are "invalid enough" to expose corner cases that have not been adequately dealt with.

For the purpose of security, input that crosses a trust boundary is often the most interesting. For example, it is more important to fuzz code that handles the upload of a file by any user than it is to fuzz the code that parses a configuration file that is accessible only to a privileged user." – Wikipedia

TARGET

JavaScriptCore is a WebKit's JavaScript engine. WebKit is a browser engine used in Apple's Safari browser as well as in several other browsers. As described by Apple, JavaScriptCore Framework provides the ability to evaluate JavaScript programs from Swift, Objective-C, and C-based apps. You can also use JavaScriptCore to insert custom objects to the JavaScript environment.

Web Browsers are present on all desktop operating systems and mobile platforms, making them an ideal target for attackers. JavaScriptCore is especially interesting from an attacker's point of view. The exploitation of a vulnerability in the framework provides a powerful exploitation primitive, allowing an attacker to compromise a Web Browser and possibly use another vulnerability Sandbox escape, potentially compromising the host system.

Finding vulnerabilities in JavaScript engines is not a trivial task. Software authors/companies and many security researchers are looking for vulnerabilities in Web browsers, making finding vulnerabilities hard and requiring a lot of research, patience, and novel approaches.

Static analysis of such a complex engine is quite complicated; therefore, an automated fuzzing process is the most used approach.

There is a lot of documentation out there focusing on many different fuzzers; however, in this writing, we'll focus on one of the most popular ones, **AFL (American Fuzzy Lop).** From a high-end picture, it's fork **American Fuzzy Lop plus plus (AFL++)**, that can be found on the following link: **https://github.com/AFLplusplus/AFLplusplus** and a custom mutator named **Superion Mutator for AFLPlusPlus**.

SETUP

In this whitepaper, we'll focus on setting up a fuzzing environment on macOS 10.15.7; even though fuzzing on macOS using AFLplusplus is not recommended fuzzing on MacOS is slow because of the unusually high overhead of **fork()**. It is possible to use **llvm-mode**; however, the recommendation is to use WebKitGTK+ on Linux machines for fuzzing.

To start, we must first clone the WebKit mirror from **https://github.com/WebKit/webkit**.

After cloning WebKit (it could take a while), we'll clone the **AFLplusplus** (link above) and build it using one of the following targets (as instructed on the project page):

- all: just the main afl++ binaries
- binary-only: everything for binary-only fuzzing: qemu_mode, unicorn_mode, libdislocator, libtokencap
- source-only: everything for source code fuzzing: llvm_mode, libdislocator, libtokencap
- distrib: everything (for both binary-only and source code fuzzing)
- man: creates simple man pages from the help option of the programs
- install: installs everything you have compiled with the build options above
- clean: cleans everything compiled, not downloads (unless not on a checkout)
- deepclean: cleans everything including downloads
- code-format: format the code, do this before you commit, and send a PR, please!
- tests: runs test cases to ensure that all features are still working as they should
- unit: perform unit tests (based on cmocka)
- help: shows these build options

For this setup, we can use **make all**.

AFLplusplus supports custom mutators that enhance and alter the mutation strategies of AFLplusplus. For this specific test, we will use Superion **Mutator for AFLPlusPlus,** available on the following link: **https://github.com/adrian-rt/superion-mutator**.

Building Superion Mutator for AFLPlusPlus is as simple as executing the build.sh script inside the project folder; however, it requires a few path corrections and AFLPlusPlus source code alterations to build successfully.

To save time and for the sake of a quick start, we built the shared object libTreeMutation.so, and it can be downloaded from here, **https://github.com/adrian-rt/superion-mutator**.

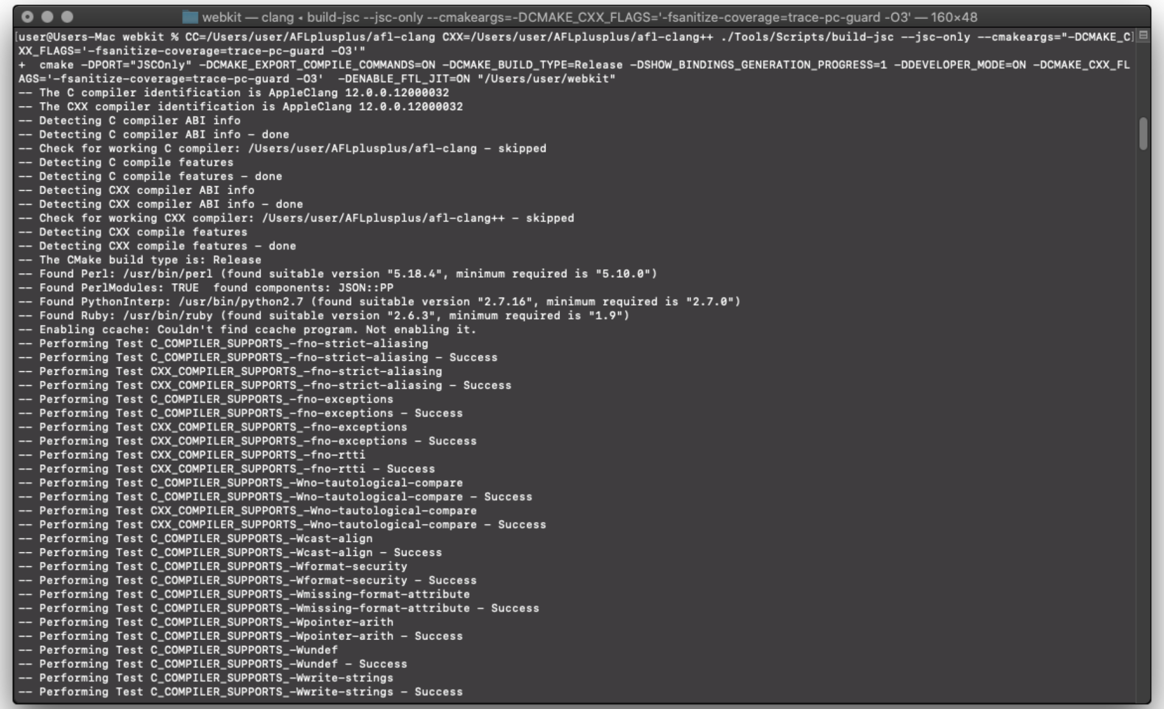
With the tools are compiled and ready to use, we still must build JavaScriptCore from the WebKit repository we cloned. Note that you must install **cmake**.

This is done by executing the following command:

CC={PATH_TO}/AFLplusplus/afl-clang CXX={PATH_TO}/AFLplusplus/afl-clang++ ./Tools/Scripts/build-jsc --jsc-only --cmakeargs="-DCMAKE_CXX_FLAGS='-fsanitize-coverage=trace-pc-guard -O3'"

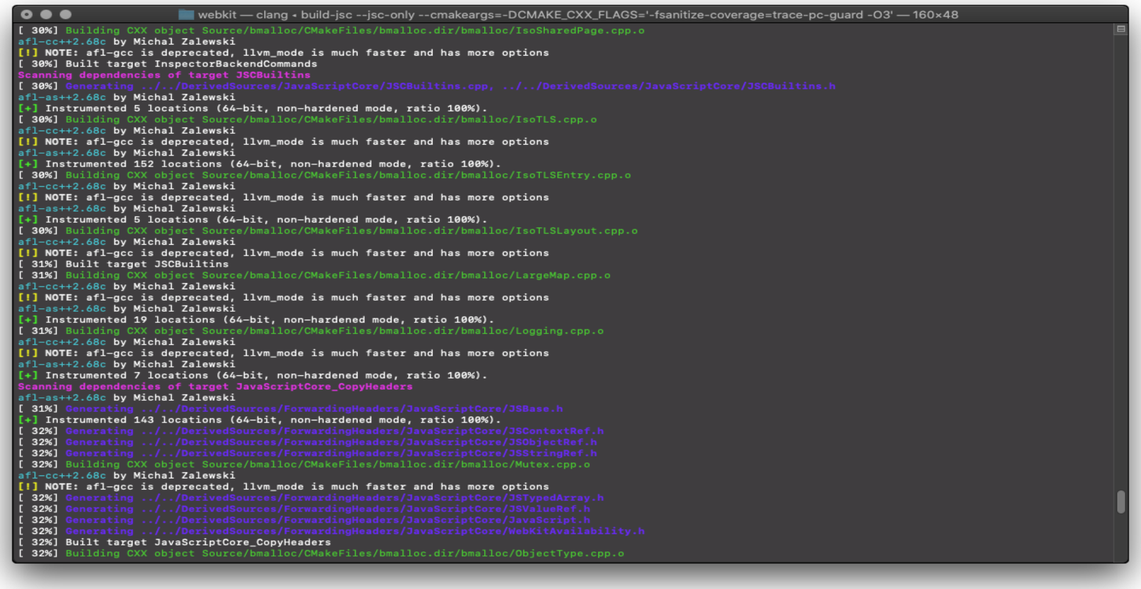
Here we will now build jsc (JavaScriptCore) only as we do not need all the WebKit functionalities.

We've also wanted to use AddressSanitizer, so we added a flag: **--cmakeargs="-DCMAKE_CXX_FLAGS='-fsanitize-coverage=trace-pc-guard -O3'"** which enables the built-in code coverage instrumentation (SanitizerCoverage) as shown in the figure below:


webkit — clang ◂ build-jsc --jsc-only --cmakeargs=-DCMAKE_CXX_FLAGS='-fsanitize-coverage=trace-pc-guard -O3' — 160×48

```
user@Users-Mac webkit % CC=/Users/user/AFLplusplus/afl-clang CXX=/Users/user/AFLplusplus/afl-clang++ ./Tools/Scripts/build-jsc --jsc-only --cmakeargs="-DCMAKE_C
XX_FLAGS='-fsanitize-coverage=trace-pc-guard -O3'"
+  cmake -DPORT="JSCOnly" -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_BUILD_TYPE=Release -DSHOW_BINDINGS_GENERATION_PROGRESS=1 -DDEVELOPER_MODE=ON -DCMAKE_CXX_FL
AGS='-fsanitize-coverage=trace-pc-guard -O3'  -DENABLE_FTL_JIT=ON "/Users/user/webkit"
-- The C compiler identification is AppleClang 12.0.0.12000032
-- The CXX compiler identification is AppleClang 12.0.0.12000032
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Users/user/AFLplusplus/afl-clang - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Users/user/AFLplusplus/afl-clang++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- The CMake build type is: Release
-- Found Perl: /usr/bin/perl (found suitable version "5.18.4", minimum required is "5.10.0")
-- Found PerlModules: TRUE  found components: JSON::PP
-- Found PythonInterp: /usr/bin/python2.7 (found suitable version "2.7.16", minimum required is "2.7.0")
-- Found Ruby: /usr/bin/ruby (found suitable version "2.6.3", minimum required is "1.9")
-- Enabling ccache: Couldn't find ccache program. Not enabling it.
-- Performing Test C_COMPILER_SUPPORTS_-fno-strict-aliasing
-- Performing Test C_COMPILER_SUPPORTS_-fno-strict-aliasing - Success
-- Performing Test CXX_COMPILER_SUPPORTS_-fno-strict-aliasing
-- Performing Test CXX_COMPILER_SUPPORTS_-fno-strict-aliasing - Success
-- Performing Test C_COMPILER_SUPPORTS_-fno-exceptions
-- Performing Test C_COMPILER_SUPPORTS_-fno-exceptions - Success
-- Performing Test CXX_COMPILER_SUPPORTS_-fno-exceptions
-- Performing Test CXX_COMPILER_SUPPORTS_-fno-exceptions - Success
-- Performing Test CXX_COMPILER_SUPPORTS_-fno-rtti
-- Performing Test CXX_COMPILER_SUPPORTS_-fno-rtti - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wno-tautological-compare
-- Performing Test C_COMPILER_SUPPORTS_-Wno-tautological-compare - Success
-- Performing Test CXX_COMPILER_SUPPORTS_-Wno-tautological-compare
-- Performing Test CXX_COMPILER_SUPPORTS_-Wno-tautological-compare - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wcast-align
-- Performing Test C_COMPILER_SUPPORTS_-Wcast-align - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wformat-security
-- Performing Test C_COMPILER_SUPPORTS_-Wformat-security - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wmissing-format-attribute
-- Performing Test C_COMPILER_SUPPORTS_-Wmissing-format-attribute - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wpointer-arith
-- Performing Test C_COMPILER_SUPPORTS_-Wpointer-arith - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wundef
-- Performing Test C_COMPILER_SUPPORTS_-Wundef - Success
-- Performing Test C_COMPILER_SUPPORTS_-Wwrite-strings
-- Performing Test C_COMPILER_SUPPORTS_-Wwrite-strings - Success
```

Now be patient. Building JavaScriptCore can take a while.



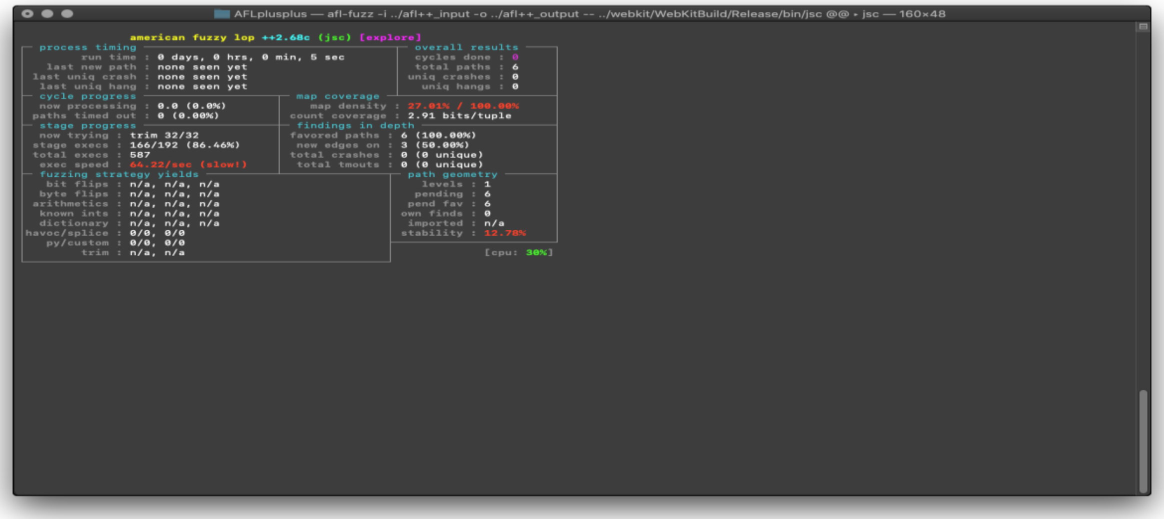
```
webkit — clang ◂ build-jsc --jsc-only --cmakeargs=-DCMAKE_CXX_FLAGS='-fsanitize-coverage=trace-pc-guard -O3' — 160×48
[ 30%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/IsoSharedPage.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
[ 30%] Built target InspectorBackendCommands
Scanning dependencies of target JSCBuiltins
[ 30%] Generating ../../DerivedSources/JavaScriptCore/JSCBuiltins.cpp, ../../DerivedSources/JavaScriptCore/JSCBuiltins.h
afl-as++2.68c by Michal Zalewski
[+] Instrumented 5 locations (64-bit, non-hardened mode, ratio 100%).
[ 30%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/IsoTLS.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
afl-as++2.68c by Michal Zalewski
[+] Instrumented 152 locations (64-bit, non-hardened mode, ratio 100%).
[ 30%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/IsoTLSEntry.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
afl-as++2.68c by Michal Zalewski
[+] Instrumented 5 locations (64-bit, non-hardened mode, ratio 100%).
[ 30%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/IsoTLSLayout.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
[ 31%] Built target JSCBuiltins
[ 31%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/LargeMap.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
afl-as++2.68c by Michal Zalewski
[+] Instrumented 19 locations (64-bit, non-hardened mode, ratio 100%).
[ 31%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/Logging.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
afl-as++2.68c by Michal Zalewski
[+] Instrumented 7 locations (64-bit, non-hardened mode, ratio 100%).
Scanning dependencies of target JavaScriptCore_CopyHeaders
afl-as++2.68c by Michal Zalewski
[ 31%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JSBase.h
[+] Instrumented 143 locations (64-bit, non-hardened mode, ratio 100%).
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JSContextRef.h
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JSObjectRef.h
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JSStringRef.h
[ 32%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/Mutex.cpp.o
afl-cc++2.68c by Michal Zalewski
[!] NOTE: afl-gcc is deprecated, llvm_mode is much faster and has more options
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JSTypedArray.h
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JSValueRef.h
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/JavaScript.h
[ 32%] Generating ../../DerivedSources/ForwardingHeaders/JavaScriptCore/WebKitAvailability.h
[ 32%] Built target JavaScriptCore_CopyHeaders
[ 32%] Building CXX object Source/bmalloc/CMakeFiles/bmalloc.dir/bmalloc/ObjectType.cpp.o
```

Once built, we are ready to begin fuzzing WebKit's JavaScriptCore, but before we embark on this journey, we must first have initial test cases.

Test cases can be various PoC's for previously disclosed JavaScript engine's vulnerabilities.

Once we have the test cases, we can start fuzzing by executing the command

AFL_CUSTOM_MUTATOR_ONLY=1 AFL_CUSTOM_MUTATOR_LIBRARY=custom_mutators/superion-mutator/libTreeMutation.so ./afl-fuzz -i ../input -o ../output -- ../webkit/WebKitBuild/Release/bin/jsc @@ as shown in the following figure:



```
AFLplusplus — afl-fuzz -i ../afl++_input -o ../afl++_output -- ../webkit/WebKitBuild/Release/bin/jsc @@ ◂ jsc — 160×48

                american fuzzy lop ++2.68c (jsc) [explore]
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 5 sec       │  cycles done : 0     │
│   last new path : none seen yet                     │  total paths : 6     │
│ last uniq crash : none seen yet                     │ uniq crashes : 0     │
│  last uniq hang : none seen yet                     │   uniq hangs : 0     │
├─ cycle progress ─────────────────┬─ map coverage ───┴──────────────────────┤
│  now processing : 0.0 (0.0%)     │    map density : 27.01% / 100.00%      │
│ paths timed out : 0 (0.00%)      │ count coverage : 2.91 bits/tuple       │
├─ stage progress ─────────────────┼─ findings in depth ─────────────────────┤
│  now trying : trim 32/32         │ favored paths : 6 (100.00%)            │
│ stage execs : 166/192 (86.46%)   │  new edges on : 3 (50.00%)             │
│ total execs : 587                │ total crashes : 0 (0 unique)           │
│  exec speed : 64.22/sec (slow!)  │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ────────┴───────────────┬─ path geometry ────────┤
│   bit flips : n/a, n/a, n/a                      │    levels : 1          │
│  byte flips : n/a, n/a, n/a                      │   pending : 6          │
│ arithmetics : n/a, n/a, n/a                      │  pend fav : 6          │
│  known ints : n/a, n/a, n/a                      │ own finds : 0          │
│  dictionary : n/a, n/a, n/a                      │  imported : n/a        │
│havoc/splice : 0/0, 0/0                           │ stability : 12.78%     │
│   py/custom : 0/0, 0/0                           ├────────────────────────┘
│        trim : n/a, n/a                           │          [cpu: 30%]
└──────────────────────────────────────────────────┘
```

TAKE AWAY

We covered only a small portion of the features and possibilities of the mentioned fuzzers.

It is important to do further research on the target, test cases, and the fuzzer and all its options and possibilities that could speed up the execution.

While this whitepaper does not present any novel research, it should serve as an introduction to fuzzing WebKit's JavaScript engine.

References:

- https://saelo.github.io/papers/thesis.pdf
- https://saelo.github.io/presentations/offensivecon_19_fuzzilli.pdf
- https://github.com/googleprojectzero/fuzzilli
- https://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Dominiak-Efficient-Approach-to-Fuzzing-Interpreters-wp.pdf
- https://github.com/Microsvuln/Awesome-AFL

Additional resources:

- https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/
- https://blog.doyensec.com/2020/09/09/fuzzilli-jerryscript.html
- https://www.techrepublic.com/article/fuzzing-fuzz-testing-tutorial-what-it-is-and-how-can-it-improve-application-security/