ICUFuzzer: Fuzzing ICU Library for Exploitable Bugs in Multiple Software

Kun Yang[1,2(✉)], Yuan Deng[3], Chao Zhang[1,2], Jianwei Zhuge[1,2], and Haixin Duan[1,2]

[1] Tsinghua University, Beijing, China
k-yang14@mails.tsinghua.edu.cn
[2] Tsinghua National Laboratory for Information Science and Technology, Beijing, China
[3] Ant-financial Light-Year Security Lab, Hangzhou, China

Abstract. Software is usually built on top of shared libraries. Vulnerabilities that lie in those dependencies may have huge impact on multiple software. ICU (International Components for Unicode) is one of the most widely used common components in modern software, providing Unicode and Globalization support. ICU is used in a wide range of software from over 70 companies and organizations, including very popular software such as Chrome, Android, macOS, iOS, Windows 10, Edge, Firefox.

In this paper, we proposed a fuzzing method to discover vulnerabilities in ICU library that are reachable from upper layer application software. We also built a prototype named ICUFuzzer to uncover triggerable bugs in browsers' JavaScript Engine, with which we have detected three zero-day vulnerabilities affecting popular browsers like Chrome, Safari and Firefox. According to our further analysis, one of the bugs can be exploited to leak sensitive memory informations to bypass mitigations like ASLR and PIE.

Keywords: ICU · Fuzzing · Browser

1 Introduction

In recent years, security researchers have done some interesting vulnerability research in compromising modern software by exploiting security flaws from third party libraries. lokihardt successfully turned a buffer overflow bug in libANGLE to get remote code execution in Chrome browser in Pwn2Own 2016. Richard Zhu leveraged a bug in libvorbis to achieve code execution in Firefox in Pwn2Own 2018. Some researcher demonstrated how to use a one byte overflow in DNS library to take remote control of Chrome OS. From the cases above, we can learn

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61472209 and No. 61772308), Tsinghua University Initiative Scientific Research Program (Grant No. 20151080436), the CCF-NSFOCUS Kunpeng Award, and the Young Talent Development Program by CCF.

L. Chen et al. (Eds.): ISC 2018, LNCS 11060, pp. 67–84, 2018.
https://doi.org/10.1007/978-3-319-99136-8_4

that those shared libraries have become huge attack surfaces in modern software. Since third party libraries are shared code among multiple software, the bugs that lie in those dependencies are also shared. Bugs in fundamental libraries may have higher impact than bugs in code that is not shared. Designing new techniques for automatically finding hidden flaws in those fundamental dependencies now become important task, especially the bugs that are reachable in upper level applications.

Our research focus on a mature, widely used libraries named ICU [10]. ICU is short for International Components for Unicode, providing Unicode and Globalization support for software applications. ICU is widely portable and gives applications the same results on all platforms and between C/C++ and Java software. ICU library is used everywhere, including document processing software like OpenOffice and LibreOffice, browsers like Chrome, Safari and Firefox, operating systems like Android, macOS, iOS, Windows and Linux. ICU is integrated as a fundamental component through high level APIs in various software stack. Only security flaws in ICU that is reachable from the input through API calls could result in security impact. So our methodology is designed for finding bugs in ICU that is triggerable from the entry point of application software.

Fuzzing is a widely used technique to detect vulnerabilities in software. We designed a fuzzing based method to effectively find exploitable bugs in ICU libraries. In our method, we will do a in-depth analysis on the connection between target software and ICU library. Target software mostly use part of APIs from ICU, and does some parameter filtering before invoking. So the analysis result will help us infer the input generation rules for fuzzing, so that we can get rid of useless mutations as many as possible while fuzzing. Moreover, we combine coverage-guided fuzzing technique to improve the code coverage.

Evaluation. To evaluate the effectiveness of our design. We implemented a prototype ICUFuzzer to detect vulnerabilities in ICU that can be triggered from browsers' JavaScript engine. We have studied how the browsers implement ECMAScript Internationalization API with the underlying support of ICU and figured out how parameters are passed from the ECMAScript API to low level ICU functions, and how the browsers filter them for security concerns. With our findings, we developed a dedicated input generation engine inside the fuzzer which produces inputs that can bypass the filters. With our methodology, we significantly improve the effectiveness of fuzzing and quickly find three zero-day bugs, 2 of them can be triggered in Chrome and 1 can be triggered in Chrome, Safari, and Firefox.

Contribution. This paper makes the following contributions:

- We are the first to conduct fuzzing research on ICU library.
- We designed a fuzzer for finding bugs in ICU that is reachable from application software. By analyzing the data flow from the input in target application to the arguments passed down to ICU, we turn fuzzing target applications

into fuzzing ICU directly to achieve better performance and keep zero false positive at the same time. Coverage guided fuzzing technique is also applied here to further improve the code coverage.
– We implemented ICUFuzzer based on our fuzzer design and effectively find three zero-day bugs that are exploitable in modern browsers.

Paper Organization. The paper is organized as follows. In Sect. 2, we give brief introduction on ICU, including the attack surfaces of ICU. In Sect. 3, we use V8 JavaScript engine as an example to describe how ICU is integrated into other applications. In Sect. 4, we will introduce our system design, and how we build the prototype ICUfuzzer. In Sect. 5, we show the results of finding bugs by ICUFuzzer. In the last part of Sects. 6 and 7, we conclude our work and give a summary on other related work.

2 Preliminaries

2.1 ICU Basics

International Components for Unicode (ICU) is an open source project of mature C/C++ and Java libraries for Unicode support, software internationalization, and software globalization. It is widely portable to many operating systems and environments. It gives applications the same results on all platforms and between C, C++, and Java software. The ICU project is sponsored, supported, and used by IBM and many other companies.

Here are a few highlights of the services provided by ICU:

– Code Page Conversion: Convert text data to or from Unicode and nearly any other character set or encoding.
– Collation: Compare strings according to the conventions and standards of a particular language, region or country.
– Formatting: Format numbers, dates, times and currency amounts according the conventions of a chosen locale.
– Time Calculations: Multiple types of calendars are provided beyond the traditional Gregorian.
– Unicode Support: ICU closely tracks the Unicode standard, providing easy access to all of the many Unicode character properties.

2.2 Who Uses ICU?

ICU is used in almost every popular operation systems. In Apple macOS, ICU library is going with the default install under the name libicucore.A.dylib. In Microsoft Windows 10, ICU library is distributed in two dynamic link libraries, icuin.dll and icuuc.dll. In Ubuntu, the package of ICU contains multiple shared libraries named libicu*.so.

ICU is heavily used in document processing software like LibreOffice, OpenOffice, PDF Box and products from Adobe. Document processing software

have been large attack surfaces used in real world APT attacks for years. Document exploits are usually malformed documents distributed by email phishing. So exploitable bugs in ICU are possible to be abused in traditional document attack scenario.

ICU is also used in trending application software like mobile operating systems, IoT devices, and smart cars. ICU can be found in Android and iOS, and even in the automotive from venders like Alfa Romeo, Audi, Mercedes-Benz, BMW.

ICU is used everywhere. The work for finding bugs in such a fundamental library is critical.

2.3 Attack Surfaces in ICU

Core function of ICU that many software rely on is Unicode and internationalization support, which are encoding translation from or to Unicode, locale conversion for numbers, datas, times and currency amounts, Unicode supported regular expression, etc. All the operations above are related to encoding and format translation, which are implemented with lots of memory manipulations. The C/C++ code for handling of large amount of specifications, encodings and locales could become large attack interfaces. In our research, we focus on memory corruption bugs in ICU's C/C++ library.

3 ICU in JavaScript Engine

3.1 ECMAScript Internationalization API

The ECMAScript Internationalization API [11] is a standard that helps handle locales of dates, numbers, and currencies in JavaScript. There is a specification that defines the application programming interface for ECMAScript objects. Most of the modern browsers implement ECMAScript Internationalization API based on ICU including Chrome, Safari and Firefox.

3.2 Architecture

To explain how ICU is integrated with JavaScript engine, we take Chrome's V8 engine as an example. Figure 1 shows the implementation architecture in Chrome's JavaScript engine V8. Every JavaScript Internationalization API call will be first handled by some internal JavaScript code and runtime engine written in C++, and then the execution goes down to the ICU library.

So JavaScript Internationalization API calls can be the surfaces of browser attacks. If malformed arguments of Internationalization API calls are passed down to ICU, bugs in ICU will be triggered. However, according to our analysis, not arbitrary arguments can be passed directly to functions in ICU. There are security checks or filters in V8, which prevents some ICU bugs to be triggered. Our fuzzer will only focus on reachable bugs.

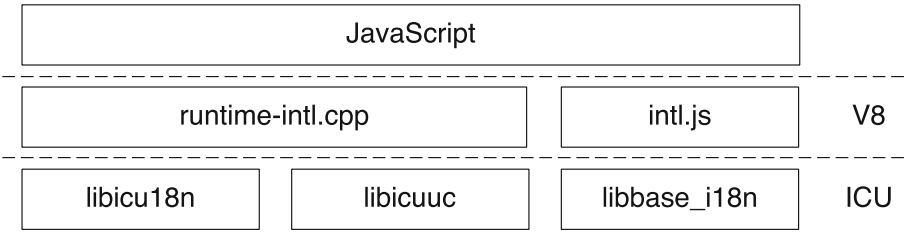

Fig. 1. Internationalization API architecture in V8

3.3 JavaScript Internationalization API

The Intl object [12] is the namespace for the ECMAScript Internationalization API, which provides language sensitive string comparison, number formatting, and date and time formatting. The Intl object have 4 properties, which are constructors for `Collator`, `NumberFormat`, `DateTimeFormat` and `PluralRules` objects:

- The `Intl.Collator` object is a constructor for collators, objects that enable language sensitive string comparison.
- The `Intl.DateTimeFormat` object is a constructor for objects that enable language-sensitive date and time formatting.
- The `Intl.NumberFormat` object is a constructor for objects that enable language sensitive number formatting.
- The `Intl.PluralRules` object is a constructor for objects that enable plural sensitive formatting and plural language rules.

The objects above are core structures in Internationalization API. The code snippet below demonstrates the character comparison between "ä" in German and "a" in English by creating an `Intl.Collator` object and calling its method `compare`.

```
var options = { sensitivity: 'base' };
new Intl.Collator('de',options).compare('ä', 'a');
// 0
```

The code snippet below illustrates the date and time formatting by creating an `Intl.DateTimeFormat` object and calling its method `format`.

```
var date = new Date(Date.UTC(2012, 11, 20, 3, 0, 0));
// request a weekday along with a long date
var options = { weekday: 'long', year: 'numeric',
                month: 'long', day: 'numeric' };
new Intl.DateTimeFormat('de-DE', options).format(date);
// "Donnerstag, 20. Dezember 2012"
```

The code snippet below illustrates the currency formatting by creating an `Intl.NumberFormat` object and calling its method `format`.

```
var number = 123456.789;
var options = {style:'currency',currency:'EUR' };
// request a currency format
new Intl.NumberFormat('de-DE', options).format(number);
// 123.456,79 €
```

Table 1 summarized all the constructors and methods in JavaScript Internationalization API. We can see all the constructors receive two optional arguments: `locales` and `options`.

Table 1. Constructors and methods in JavaScript internationalization API

Objects/Constructors	Methods
Intl.Collator(locales, options)	Intl.Collator.prototype.resolvedOptions()
Intl.DateTimeFormat(locales, options)	Intl.DateTimeFormat.prototype.format(date)
	Intl.DateTimeFormat.prototype.formatToParts(date)
	Intl.DateTimeFormat.prototype.resolvedOptions()
Intl.NumberFormat(locales, options)	Intl.NumberFormat.prototype.format(number)
	Intl.NumberFormat.prototype.formatToParts(number)
	Intl.NumberFormat.prototype.resolvedOptions()
Intl.PluralRules(locales, options)	Intl.PluralRules.prototype.resolvedOptions()
	Intl.PluralRules.prototype.select(number)

Locales Argument. The `locales` argument must be either a string holding a BCP 47 language tag [13], or an array of such language tags. A BCP 47 language tag defines a language and minimally contains a primary language code. In its most common form it can contain, in order: a language code, a script code, and a country or region code, all separated by hyphens. The subtags identifying languages, scripts, countries (regions), and (rarely used) variants in BCP 47 language tags can be found in the IANA Language Subtag Registry [10]. While the tag is not case sensitive, it is recommended to use title case for script code, upper case for country and region codes and lower case for everything else. The following are examples of `locales`:

- "hi": Hindi (primary language).
- "de-AT": German as used in Austria (primary language with country code).
- "zh-Hans-CN": Chinese written in simplified characters as used in China (primary language with script and country codes).

BCP 47 also allows for extensions. JavaScript internationalization functions use the "u" (Unicode) extension, which can be used to request additional customization of `Collator`, `NumberFormat`, or `DateTimeFormat` objects. The Unicode extension uses additional keys as subtags.

Let's take the constructor `Intl.Collator` as an example. `Collator` supports 3 Unicode extension keys, `co`, `kn`, and `kf`. These keys can be used in the `locales`,

and `locales` can be passed to the constructor to customize the `Collator` object:

- `co` is for variant collations for certain locales. Possible values include: `"big5-han"`, `"dict"`, `"direct"`, `"ducet"`, `"gb2312"`, `"phonebk"`, `"phonetic"`, `"pinyin"`, etc.
- `kn` specifies whether numeric collation should be used, such that "1" < "2" < "10". Possible values are `"true"` and `"false"`.
- `kf` specifies whether upper case or lower case should sort first. Possible values are `"upper"`, `"lower"`, or `"false"` (use the locale's default).

More details about Unicode extension keys for other objects can be found in MDN web docs [12]. Table 2 summarizes extension keys and possible values. Below are some example `locales` that use Unicode extension keys:

- `"de-DE-u-co-phonebk"`: Use the phonebook variant of the German sort order, which expands umlauted vowels to character pairs: ä → ae, ö → oe, ü → ue.
- `"en-GB-u-ca-islamic"`: Use British English with the Islamic (Hijri) calendar, where the Gregorian date 14 October, 2017 is the Hijri date 24 Muharram, 1439.

Table 2. Unicode extensions keys and values in Locales argument

Constructor	Unicode extension key	Possible values
`Collator`	`co`	"big5han", "dict", "direct", "ducet", "gb2312", "phonebk", "phonetic", "pinyin", etc.
	`kn`	"true", "false"
	`kf`	"upper", "lower", "false"
`DateTimeFormat`	`nu`	"arab", "arabext", "bali", "beng", "deva", "fullwide", "gujr", "guru", "hanidec", etc.
	`ca`	"buddhist", "chinese", "coptic", "ethioaa", "ethiopic", "gregory", "hebrew", "indian", "islamic", "islamicc", etc.
	`hc`	"h11", "h12", "h23", "h24"
`NumberFormat`	`nu`	"arab", "arabext", "bali", "beng", "deva", "fullwide", "gujr", "guru", "hanidec", etc.

Options Argument. The `options` argument must be an object with properties that vary between constructors and functions. Properties in the `options` argument are like optional configurations. Different constructors and functions require different properties. For example, the constructor `Intl.DateTimeFormat` supports some properties as below:

- `localeMatcher` specifies the locale matching algorithm to use. Possible values are "lookup" and "best fit"; the default is "best fit". This property is supported by all language sensitive constructors and functions.
- `timeZone` specifies the time zone to use, such as `"Asia/Shanghai"`, `"Asia/Kolkata"`, `"America/New_York"`.
- `hour12` specifies Whether to use 12-h time (as opposed to 24-h time). Possible values are true and false; the default is locale dependent.

Detailed information on properties and functions can be retrieved from MDN web docs [12].

3.4 Data Flow from JavaScript to ICU

In previous section, we have gone through the entry level data structures and methods in Javascript International API. In this part, we will dive deep into the implementation of V8 engine to understand how input arguments are handled and passed to ICU.

We will follow the execution of constructor and methods of `Intl.DateTimeFormat` to understand the argument data flow. Figure 2 depicts the basic

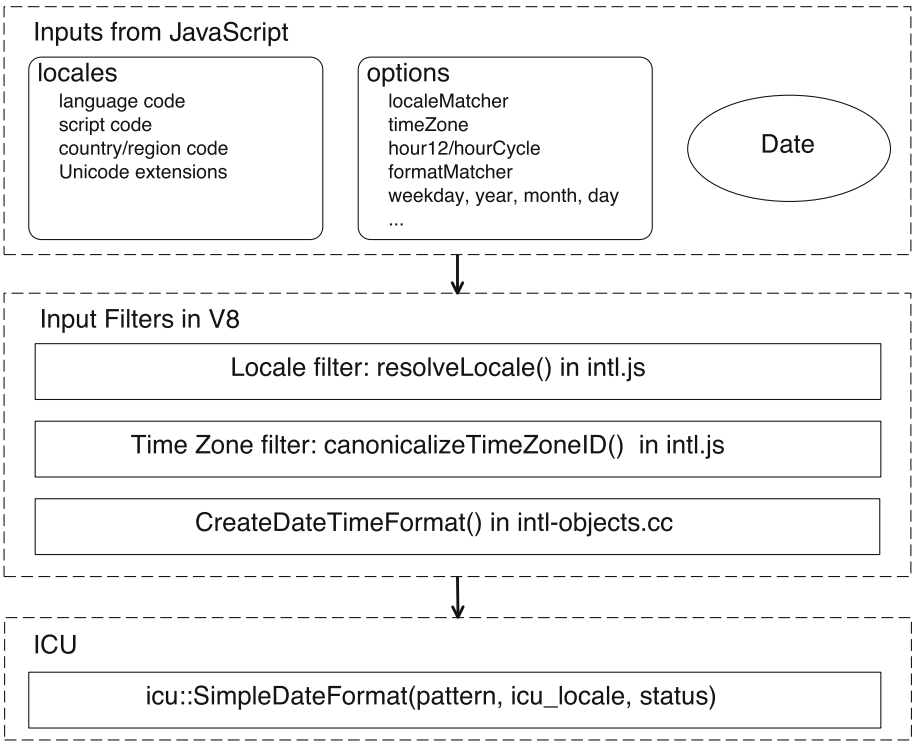

Fig. 2. Date and Time related International API's Input Filtering Process in V8

procedure. The inputs for `Intl.DateTimeFormat`' are `locales`, `options` and `Date`. `locales` will be resolved in function `resolveLocale()` and filtered. Then time zone information will be extracted from `options` and filtered in `canonicalizeTimeZoneID()`. Both filters are implemented in internal JavaScript code intl.js.

Locale Filtering. For `locale` filtering in `resolveLocale()`, the following steps will be taken:

1. Canonicalize the language code in function `canonicalizeLanguageTag()`
   (a) Call function `isStructuallyValidLanguageTag()`, to match the `locale` by three regular expressions: `LANGUAGE_TAG_RE`, `LANGUAGE_VARIANT_RE` and `LANGUAGE_SINGLETON_RE`
   (b) check if `locale` conform to strict BCP47 rules by performing a pair of invertible operations: calling `uloc_forLanguageTag()` and `uloc_toLanguageTag()`
2. Split the Unicode extensions from `locale`
3. Check extension keys extracted from `locale` by matching a pre-initialized key list

Options Filtering. Just like `locale`, V8 will also filter the argument `options` before passing it to ICU. In the constructor `Intl.DateTimeFormat`, the property `timeZone` in argument `options` will be filtered in function `canonicalizeTimeZoneID()` by following steps:

1. Use regular expression to check if `timeZone` string matches the format of *Area/Location(/Location)**
2. Convert the location string to be titlecased by function `toTitleCaseTimezoneLocation()`. For example, convert "bueNos_airES" to "Buenos_Aires".

4 System Design

4.1 Design Goals

In Sect. 3, we study how ICU is integrated inside V8 engine. Most of modern application software like browsers are designed and implemented in a hierarchical structure, which depends on a lot of fundamental libraries in the low level. ICU is one of the critical components and has not received much attention on security aspect.

In this paper, we aim at designing a fuzzing methodology to uncover the bugs in ICU library. It is worth emphasizing that the ICU bugs that we are fuzzing for should be reachable from target applications. Our fuzzer is designed for hunting bugs that can affect specific application. The reason is that, if we consider a library separately, the threat model may not be very clear. Libraries are only responsible for exposing dozens of APIs, and library developers may

know little on how these interfaces will be used in the applications. So if we apply bug finding in ICU in the context of application software, the results will bring more practical value.

Research on general fuzzing techniques has been lasted for long time and already made very good progress. In our research, we are going to apply the in-depth understanding on ICU into the fuzzer to achieve better performance and results.

4.2 Design Challenges

Fuzzing ICU for exploitable bugs in target application could be carried out in two different ways. One is fuzzing the application directly. By mutating and feeding the input that is ICU related to the target software, the crashes you get are real bugs that will surely affect the application. However, according to our analysis on V8, there will be a middle layer between the input layer and ICU that is always checking or filtering the test cases generated by the fuzzer. So the mutation will be struggled in surviving the filters, large amount of computation resources in fuzzing will be wasted in finding paths in the application that can reach ICU. Meanwhile, compared with ICU library, application is more complex software like browsers and pdf readers. Fuzzing ICU in the context of whole application is slow. To conclude, this method has an advantage of 0 false negative, but has low efficiency.

The other method is fuzzing ICU APIs directly. By figuring out which APIs target application relies on, fuzzer can skim the target software and feed mutated arguments directly to ICU APIs. The second method also has a drawback that even if you find a test case that can crash ICU, the bug could not be reproduced in the context of target application. The input will go into the application first and be filtered before being passed to the low level's library. This idea can help improve the fuzzing throughput, but will have very high false positive.

In Fig. 3, it shows the different fuzzing points between two methods discussed above. The challenge we are facing here is that we cannot have low false negative and high fuzzing throughput at the same time.

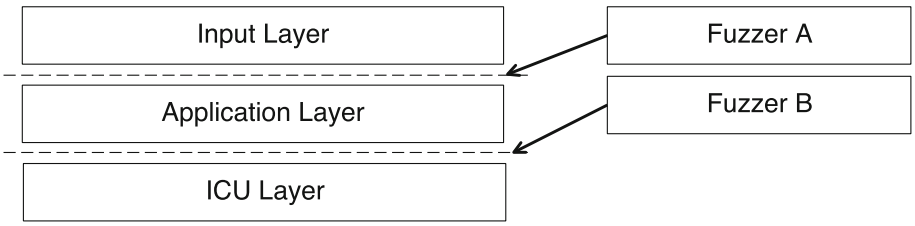

Fig. 3. Different Fuzzing Point Between Two Methods

4.3 System Architecture

To overcome the drawbacks in two fuzzing methods discussed above, and keep the good parts for both, we can fuzz the ICU API directly while embedding the filters in target application into the fuzzer to make sure there is no false positive. By analyzing ICU and its depending software, we can obtain the filtering rules like what we have done in Sect. 3 for browsers. Then by embedding the rules inside our fuzzer, we could filter the invalid test cases before feeding them to ICU. In the Fig. 4 below, it illustrates the basic idea of our fuzzer. We call this kind of fuzzer context aware fuzzer, because by applying the filtering rules in the application, the fuzzer is granted with the knowledge from the target software. Every test case will be checked inside fuzzer before going to ICU, and there is no need to run the entire application while fuzzing.

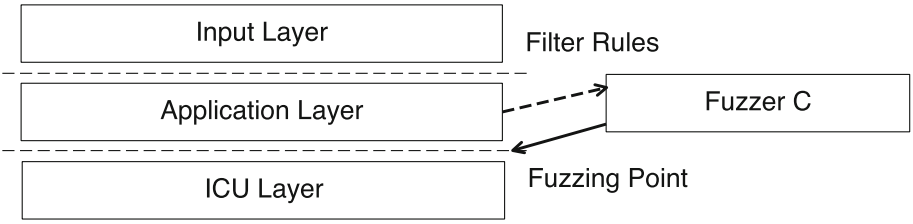

Fig. 4. Context aware ICU fuzzer

In our fuzzer, we can further improve the effectiveness of test cases generation by applying grey-box fuzzing method. We introduced coverage guided fuzzing technique here. In Fig. 5, we can see all of the internal modules for the ICU fuzzer. The *Executor* always get new test case from the *Mutator*, then pass it

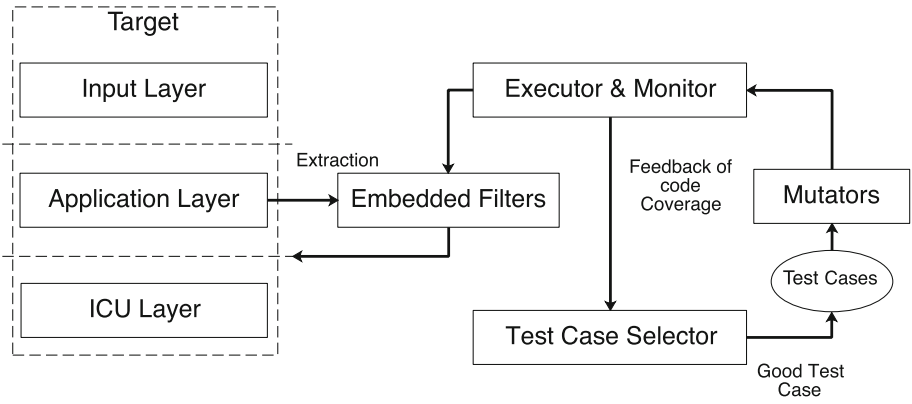

Fig. 5. Context aware ICU fuzzer combined with coverage guided fuzzing

to the *Embedded Filters* extracted from target application. The filter will stop the execution if the test case is not valid, other than the execution will go to the ICU API for testing. *Executor* should be able to retrieve the code coverage information and decide if this test case should be further mutated or abandoned. Good test cases will be remained and the execution will be guided to discover new branches in ICU library.

Executor and Embedded Filters. The argument filters in the applications are usually implemented for filtering `locale` strings and various properties in `options`. According to Sect. 3, we can see in the browser case, most of these filters are regular expressions. So we can collect all the rules and program them in the fuzzer as embedded filters. For the test inputs that can bypass the filters, we pass them to native ICU calls.

Mutation. Different ICU APIs have different arguments. In order to design a mutation method that is able to adapt to different number and types of arguments. We use a popular mechanism supported in almost every programing language to unify the input data for different ICU APIs—serialization and deserialization. We take byte stream as a unifying input format in fuzzing. Mutation is easy to be implemented on byte stream, e.g. bit flips, byte flips, bit rotations or arithmetic operations. When a test case of byte stream is delivered to *Executor*, the input stream will first be deserialized into corresponding arguments that is compatible with ICU APIs to be tested. Figure 6 demonstrates the deserialization and mutation process.

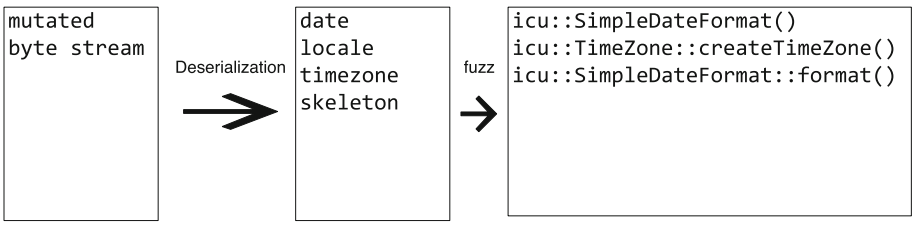

Fig. 6. Context aware ICU fuzzer combined with coverage guided fuzzing

4.4 ICUFuzzer for JavaScript Engine

To evaluate our design, we implemented a prototype named ICUFuzzer to fuzz for exploitable bugs in browsers. We use libFuzzer [19] as underlying support for coverage-guided fuzzing. We focus on fuzzing International APIs—constructors and methods of 4 core objects: `Collator`, `DateTimeFormat`, `NumberFormat` and `PluralRules`, as listed in Table 1. Let's take `DateTimeFomat` and `NumberFormat` as two examples to show how we map the upper level APIs to low level ICU APIs.

Fuzzing `Intl.DateTimeFomat`. In JavaScript, methods from the object `DateTimeFomat` are invoked in the following manner:

1. Create an object of `Intl.DateTimeFormat` by the constructor, specifying `locales` and `options`.
2. Use the object initialized in the previous step to call `Format(date)` or `formatToParts(date)` to format `date` into specified `locales` and `options`.

All the arguments specified in JavaScript will be filtered in V8 and passed to APIs in ICU. After passing down to the ICU part, three methods will be invoked:

1. `icu::SimpleDateFormat()` creates an object `SimpleDateFormat` in ICU from `locales` and properties in `options`.
2. `icu::TimeZone::createTimeZone()` creates the `TimeZone` object information in ICU from `timeZone` property in `options`.
3. `icu::SimpleDateFormat::format()` is responsible for arbitrary date formatting according to the configuration.

According to our design, ICUFuzzer will directly fuzz the ICU functions (Figs. 7 and 8).

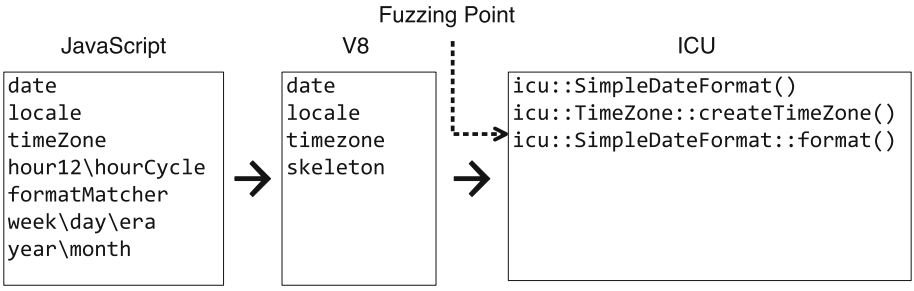

Fig. 7. Data flow in `Intl.DateTimeFomat`

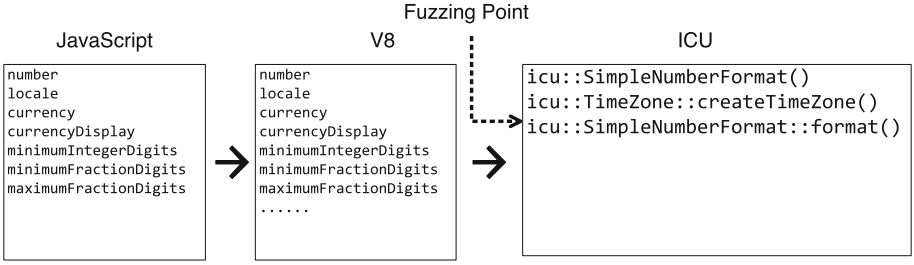

Fig. 8. Data flow in `Intl.NumberFormat`

Fuzzing `Intl.NumberFormat`. In JavaScript, methods from the object `DateTimeFomat` are invoked in the following procedure:

1. Create an object of `Intl.NumberFormat` by the constructor, specifying `locales` and `options`.
2. Use the object initialized in the previous step to call `Format(date)` or `formatToParts(date)` to format `number` into specified `locales` and `options`.

All the arguments specified in JavaScript will be filtered in V8 and passed to APIs in ICU. After passing down to the ICU part, three methods will be invoked:

1. `icu::SimpleNumberFormat()` creates an object `SimpleNumberFormat` in ICU from `locales` and properties in `options`.
2. `icu::TimeZone::createTimeZone()` creates the `TimeZone` object information in ICU from `timeZone` property in `options`.
3. `icu::SimpleNumberFormat::format()` is responsible for arbitrary number formatting according to the configuration.

Filters in V8. According to the analysis result in Sect. 2, there are `locale` filter and `options` filter implemented in V8's Intl.js. We extracted all the filtering rules for `locale` as multiple regular expressions as below.

```
regex simple_re("^[a-z]{2,3}$", std::regex::ECMAScript);
regex singleton_re("^([0-9]|[A-WY-Za-wy-z])$", std::regex::ECMAScript |
    std::regex::icase);
regex variant_re("^(([a-zA-Z]|[0-9]){5,8}|([0-9]([a-zA-Z]|[0-9]){3}))$",
    std::regex::ECMAScript | std::regex::icase);
regex langtag_re("^(([a-zA-Z]{2,3}(-([a-zA-Z]{3}(-[a-zA-Z]{3}){0,2}))?|[a-
    zA-Z]{4}|[a-zA-Z]{5,8})(-([a-zA-Z]{4}))?(-([a-zA-Z]{2}|[0-9]{3}))
    ?(-(([a-zA-Z]|[0-9]){5,8}|([0-9]([a-zA-Z]|[0-9]){3})))*(-(([0-9]|[A-WY
    -Za-wy-z])(-([a-zA-Z]|[0-9]){2,8})+))*(-(x(-([a-zA-Z]|[0-9]){1,8})+))
    ?|(x(-([a-zA-Z]|[0-9]){1,8})+)|((en-GB-oed|i-ami|i-bnn|i-default|i-
    enochian|i-hak|i-klingon|i-lux|i-mingo|i-navajo|i-pwn|i-tao|i-tay|i-
    tsu|sgn-BE-FR|sgn-BE-NL|sgn-CH-DE)|(art-lojban|cel-gaulish|no-bok|no-
    nyn|zh-guoyu|zh-hakka|zh-min|zh-min-nan|zh-xiang)))$", std::regex::
    ECMAScript | std::regex::icase);
regex any_ext_re("-[a-z0-9]{1}-.*", std::regex::ECMAScript);
regex uni_ext_re("-u(-[a-z0-9]{2,8})+", std::regex::ECMAScript);
regex locale_re("^([a-z]{2,3})-([A-Z][a-z]{3})-([A-Z]{2})$", std::regex::
    ECMAScript);
```

For `options`, we also listed the checking rules in V8 for different properties in Table 3.

Table 3. Part of extracted rules in V8's `options` filter.

Property	Type	Filtering rules for value
Weekday	string	narrow: 'EEEEE', short: 'EEE', long: 'EEEE'
Era	string	narrow: 'GGGGG', short: 'GGG', long: 'GGGG'
Year	string	2-digit: 'yy', numeric: 'y'
Month	string	2-digit: 'MM', numeric: 'M', narrow: 'MMMMM', short: 'MMM', long: 'MMMM'
Day	string	2-digit: 'dd', numeric: 'd'

The filters above are embedded into ICUFuzzer to get rid of potential false positive test cases, which could only crash ICU but not the browsers.

5 Evaluation

5.1 Experiment Design

To evaluate bug hunting capability of ICUFuzzer, we run it in a desktop PC that equipped with Intel i7 processor with clock speed of 2.67 GHz, with 8 Gb of RAM, and with Ubuntu 16.04. Within 10 min, we got dozens of crashes. We manually analyzed and classified the crashes, then we realized we found 3 0 days in latest ICU code base, and all of them can crash the latest Chrome browser through loading a snippet of JavaScript where we invoke JavaScript International APIs with malformed arguments. We reported to Chrome and have all the bugs fixed. We will go through the 3 bugs in the next part.

5.2 Results

CVE-2017-15422: Persian Calendar Integer Overflow. This bug affects Chrome, Safari and Firefox and exists in the code snippet below:

```
//i18n/persncal.cpp
void PersianCalendar::handleComputeFields(int32_t julianDay,
     UErrorCode &/*status*/)
{
int32_t daysSinceEpoch = julianDay - PERSIAN_EPOCH;//
year = 1 + ClockMath::floorDivide(33 * daysSinceEpoch + 3,
    12053);
.......
dayOfMonth = dayOfYear - kPersianNumDays[month] + 1; // Out
    of bound memory read
}
```

Integer overflow can happen in the expression `33 * daysSinceEpoch`, leading to an unbounded `month` value. The unbounded `month` value will be used as an index in array `kPersianNumDays`, which can be exploited to read out of bound memory. We have successfully turn the bug into a working exploit targeting

Chrome, Safari and Firefox to leak memory addresses, which means modern mitigations ASLR and PIE are bypassed. According our survey, this bug has been existed for more than 5 years.

Here is the proof of concept code that can trigger the bug in the browser.

```
var dateformatter = new Intl.DateTimeFormat("bs-Cyrl-u-ca-
    persian");
date = null;
Date.prototype["valueOf"] = function (){};//date returns NaN
d = dateformatter.formatToParts(date);
```

CVE-2017-15396: NumberingSystem::createInstance Stack Overflow. This bug only affects Chrome. Look at the code snippet below:

```
char buffer[ 96 ];
int32_t count = inLocale.getKeywordValue("numbers",buffer,
    sizeof(buffer),status);
if ( count > 0 ) {
        buffer[count] = '\0';   //count = 99
```

The integer variable `count` can exceed the size of buffer, leading to a stack out of bound write.

Proof of concept code is as below:

```
var nf = new Intl.NumberFormat('bs-u-nu-bzcu-cab-cabs-
    avnlubs-avnihu-zcu-cab-cbs-avnllubs-avnihq-zcu-cab-cbs-
    ubs-avnihu-cabs-flus-xxd-vnluy');
```

CVE-2017-15406: CanonicalizeLanguageTag Stack Overflow. This bug only affects Chrome. Look at the code snippet below:

```
char icu_result[ULOC_FULLNAME_CAPACITY];
uloc_forLanguageTag(*locale_id, icu_result,
    ULOC_FULLNAME_CAPACITY, nullptr, &error);
// localeID is not terminated with null byte
...
if (uprv_strlen(localeID) > 0) {        // overflowed
```

`localeID` doesn't need to be null terminated, so the result of `uprv_strlen(localeID)` could be oversized, leading to a stack out of bound write.

Proof of concept code is as below:

```
var dateti1 = new Intl.DateTimeFormat("iw-up-a-caiaup-araup-
    ai-pdu-sp-bs-up-arscna-zeieiaup-araup-arscia-rews-us-up-
    arscna-zeieiaup-araup-arsciap-arscna-zeieiaup-araup-
    arscie-u-sp-bs-uaup-arscia");
```

6 Related Work

Fuzzing was originally introduced as one of several tools to test UNIX utilities [1]. Since then, much work has been devoted to improving general fuzzing technique. Guided fuzzing is proposed to direct fuzzers toward specific types of vulnerabilities. One of the typical design for guided fuzzing is selectively choosing optimal test cases [2,3]. Dowser [2] and BuzzFuzz [4] applies taint-tracking to analyze the relations between test cases and code regions in target software. Flayer [5] allows an auditor to skip complex checks in the target application to improve the code coverage of fuzzing. Similarly, Taintscope [6] uses a checksum detection algorithm to remove checksum code from applications. Symbolic execution is also applied to fuzzing to gain maximal code coverage [7–9]. These approaches reply on symbolic execution to generate inputs that will take new code paths.

There are many other research focus on applying fuzzing in some application scenarios and making improvement by using unique characteristics in the scenarios, including our research in the paper. IntentFuzzer [15] applies dynamic fuzzing method in Android IPC mechanism to find permission leak vulnerabilities. jFuzz [16] uses a combination of concrete and symbolic execution to fuzz Java program. MTF [17] proposed a guided fuzzing strategy instead of random testing to fuzz Modbus/TCP protocol. IOTFUZZER [18] applies the knowledge from mobile applications in the fuzzer to find memory corruptions in IoT firmware.

7 Conclusion

In this paper, we proposed a fuzzing based method to discover bugs in ICU. Our method is interested in the bugs that are exploitable in the context of application software that use ICU. After we conducted a in-depth study on the implementation details of JavaScript International API, we understand the data flow from entry level input to low level ICU functions and the underlying input filtering mechanism. By applying these knowledge to the fuzzer, we improve the fuzzer to have high throughput and low false positive at the same time. We implemented a prototype named ICUFuzzer to evaluate our design. ICUfuzzer is dedicated for finding ICU bugs in browsers. By running the fuzzer, we have found three zero-day bugs in modern browsers. One of them can be exploited to leak memory information, the others can crash the browser and possibly get remote code execution.

References

1. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
2. Haller, I., Slowinska, A., Neugschwandtner, M., et al.: Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: USENIX Security Symposium, pp. 49–64 (2013)

3. Neugschwandtner, M., Milani Comparetti, P., Haller, I., et al.: The BORG: nanoprobing binaries for buffer overreads. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 87–97. ACM (2015)
4. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering, pp. 474–484. IEEE Computer Society (2009)
5. Drewry, W., Ormandy, T.: Flayer: exposing application internals. WOOT **7**, 1–9 (2007)
6. Wang, T., Wei, T., Gu, G., et al.: TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: IEEE symposium on Security and privacy (SP), pp. 497–512. IEEE (2010)
7. Caselden, D., Bazhanyuk, A., Payer, M., et al.: Transformation-aware Exploit Generation using a HI-CFG. Department of Electrical Engineering and Computer Science, California University, Berkeley (2013)
8. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: ACM Sigplan Notices, vol. 40, no. 6, pp. 213–223 (2005)
9. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
10. ICU - International Components for Unicode. http://site.icu-project.org/
11. ECMAScript Internationalization API Specification. https://www.ecma-international.org/ecma-402/1.0/
12. Document for Intl object from MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
13. RFC 5646 - Tags for Identifying Languages. https://tools.ietf.org/html/rfc5646
14. IANA Language Subtag Registry. https://www.iana.org/assignments/language-subtag-registry/language-subtag-registry
15. Yang, K., Zhuge, J., Wang, Y., et al.: IntentFuzzer: detecting capability leaks of android applications. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 531–536. ACM (2014)
16. Jayaraman, K., Harvison, D., Ganesh, V., et al.: jFuzz: a concolic whitebox fuzzer for Java (2009)
17. Voyiatzis, A.G., Katsigiannis, K., Koubias, S.: A Modbus/TCP fuzzer for testing internetworked industrial systems. In: 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–6. IEEE (2015)
18. Chen, J., Diao, W., Zhao, Q., et al.: IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing
19. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html