A Hybrid Analysis to Detect Java Serialisation Vulnerabilities

Shawn Rasheed
Massey University
Palmerston North, New Zealand
s.rasheed@massey.ac.nz

Jens Dietrich
Victoria University of Wellington
Wellington, New Zealand
jens.dietrich@vuw.ac.nz

ABSTRACT

Serialisation related security vulnerabilities have recently been reported for numerous Java applications. Since serialisation presents both soundness and precision challenges for static analysis, it can be difficult for analyses to precisely pinpoint serialisation vulnerabilities in a Java library. In this paper, we propose a hybrid approach that extends a static analysis with fuzzing to detect serialisation vulnerabilities. The novelty of our approach is in its use of a heap abstraction to direct fuzzing for vulnerabilities in Java libraries. This guides fuzzing to produce results quickly and effectively, and it validates static analysis reports automatically. Our approach shows potential as it can detect known serialisation vulnerabilities in the Apache Commons Collections library.

KEYWORDS

Java Serialisation, Program Analysis, Security Analysis

ACM Reference Format:
Shawn Rasheed and Jens Dietrich. 2020. A Hybrid Analysis to Detect Java Serialisation Vulnerabilities. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3418931

1 INTRODUCTION AND MOTIVATION

1.1 Serialisation in Java

Serialisation is the mechanism used to capture program state for persistence of runtime state or for procedure calls across process boundaries. It involves the conversion of internal runtime representations to a stream of binary or character data and back [20]. Serialisation is supported in many object-oriented languages and support for it is included in the standard libraries of most of these languages. In Java, native serialisation objects are constructed extralinguistically (i.e. bypass constructors) and object state is restored from deserialised data. Users can customise (de)serialisation by implementing the `readObject` method for a serialisable class. Java serialisation is known to be a problematic feature [2, 6, 16, 21]. There are two known weaknesses in Java's native serialisation: (1) the possibility of malformed objects as the internals of the object are exposed in the serialised stream (2) deserialisation may trigger unintended code execution with data from the input stream.

Serialisation-related vulnerabilities (e.g. CVE-2016-4000, CVE-2015-3253 [11, 14]) have been found in widely used Java libraries such as Groovy and Apache Commons Collections. These include *arbitrary code execution* [17], *expression injection* [33] and *system resource access* [13] vulnerabilities. Their impact warranted serialisation to be listed in OWASP's list of the top 10 most critical web application security risks for 2018 [1].

1.2 Detecting Serialisation Vulnerabilities

The question is how to construct effective program analyses to detect such vulnerabilities. Most static analyses, including points-to and call graph construction, are not sound [25]. Much of the unsoundness is from dynamic program behaviour due to the use of programming features such as reflection, dynamic class loading, proxies and native methods [35]. In some cases, soundness can be recovered by heavily over-approximating dynamic program behaviour, which can be costly due to loss of precision [25].

It turns out that the most well known deserialisation vulnerabilities [2] in Java use at least one of those dynamic features [34]. For instance, the Apache Commons Collections attack (CVE-2015-4852 [12]) exploits runtime reflection and the Groovy vulnerability (CVE-2015-3253 [11]) leverages Java's dynamic proxy feature. This suggests that a program where the deserialisation of an object leads to the invocation of such a feature is inherently vulnerable. For instance, if Java deserialisation (i.e., `readObject()`) triggers an execution of `Class::getMethod(..)` and `Method::invoke(..)`, and there is a flow from a field of an object in the deserialised object graph to the parameters of those methods, then this can be used easily to launch a code execution attack (e.g., via a reflective invocation of `Runtime::exec(..)` or a Denial of Service (DoS) attack (e.g., by allocating large data structures to exhaust memory, like `new ArrayList(size)`). We refer to the call sites of these features as *dynamic sinks*, because beyond these call sites it might be difficult or impossible for an analysis to reason, and deserialised data may possibly flow to a security sensitive sink method such as `Method::invoke(..)`.

To the best of our knowledge, there is only one tool (*Gadget Inspector*[3]) for detecting serialisation vulnerabilities. This tool is purely static, and it is based on call graph construction and data flow analysis to find potential vulnerabilities. It works by finding *gadget chains* - a call chain ending with an invocation of a security-sensitive method. Since the tool is based on static analysis, it can be prone to precision issues, and it requires the user to manually examine and confirm the results.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418931

[1] https://owasp.org/www-project-top-ten/
[2] https://github.com/frohoff/ysoserial
[3] https://github.com/JackOfMostTrades/gadgetinspector

To improve on the deficiencies of a purely static analysis, this paper proposes a hybrid analysis to detect potential Java serialisation vulnerabilities. The proposed method considers call sites that have reflective targets as unsafe dynamic sinks. The analysis is composed of two steps: the first step is to run a static analysis to detect call graph chains linking deserialisation call sites with dynamic sinks. We also look for data flows from the deserialised objects to the dynamic sinks via heap access paths. In the second step, we use the results from the static analysis and a dynamic technique, fuzzing, to construct actual input objects. By construction, the deserialisation of those objects triggers the invocation of dynamic sinks with user input that is read from the stream, which can be considered as tainted. While this technique is inherently unsound, our experimental validation suggests that it is able to produce objects representing malicious inputs when deserialised.

1.3 Running Example

A running example of a serialisation vulnerability is shown in Listing 1. To introduce some terminology, we refer to methods triggered by `readObject` as *trampoline methods*. For instance, in the Java collections library, `HashMap::readObject` triggers the trampoline method `Object::hashCode`. This method is triggered to recompute the state of the `HashMap` from serialised data. Source objects are serialisable objects with trampoline methods. These objects are sources for data to enter the system, and there is a potential vulnerability if that data flows to a dynamic sink [1].

The `Container` class has two fields of types `Element` and `A` (a serialisable class with concrete implementations `B` and `C`). When a stream containing a source object of type `Container` is deserialised using a `hashCode` trampoline on the `Container` object, it triggers the `getHash` method, depending on the type of the `property` field in the serialised object. If the object is of type `C` it invokes an arbitrary method specified in a string which is also obtained from the serialised object. If a program uses untrusted serialisation, and if these classes are in the class path they can be utilised by an attacker to execute a method from an arbitrary class.

```
1  public class C extends A {
2    int getHash(Element e) {
3      return e.hashCode();
4    }
5  }
6
7  public class Container implements Serializable {
8    Element e;
9    A property;
10
11   public int hashCode() {
12     return property.getHash(e);
13   }
14 }
15
16 public class Element implements Serializable {
17   String methodName;
18   String className;
19
20   public Object valueOf() {
21     Class klass = Class.forName(className);
22     Method m = klass.getMethod(methodName);
23     o = m.invoke(klass.newInstance());
24     return o;
25    }
26   public int hashCode() {
27     return valueOf().hashCode();
28   }
29 }
```

Listing 1: Java code with serialisation vulnerability.

2 ANALYSIS

The analysis runs in two steps. In the first step, the static analysis takes a Java library as input, and performs on-the-fly call-graph construction and points-to analysis with trampoline methods as entry points to obtain information about the flow of values through the heap. To compute interesting heap flows, the analysis uses constraints we describe later in this section. A heap access path has information on the objects and fields which participate in the flow of a value from a serialisable object to a parameter of a dynamic sink. In the second phase, an object is reconstructed based on these results. The trampoline method is executed with the object as an argument, and we verify whether the source-to-sink method execution path is observed. If it is not, we use the heap information as a seed for a fuzzer to explore inputs for a source-to-sink execution.

2.1 Static Analysis

The static analysis examines the given library to detect the following:

(1) a source object with a trampoline method that starts a call chain to a dynamic sink
(2) a sink object that is reachable via a heap access path (see 2.1.3) from the source object, and to which a parameter of the dynamic sink points to.

In Listing 1, the source object is an instance of `Container` as it has a trampoline method, `hashCode`, in the call graph. The dynamic sinks are the call sites for `Class::forName`, `Class::getMethod` or `Method::invoke`. The field `className` aliases with a parameter for `Class::forName`, and since that field is in an access path from the source object, we consider the call site with the target `Class::forName` to be a dynamic sink as well as a tainted call site.

2.1.1 Points-to graph. In order to discuss the flow of values we must consider the points-to graph [32], which consists of abstract heap objects and variables as nodes and pointer-manipulating statements as edges. Pointer-manipulating statements are new object allocations, local assignments, interprocedural assignments through return statements and parameter bindings at call sites, and loads or stores from an object's fields. A variable points to an abstract location if there is a path from that variable to the location in the graph.

2.1.2 Native heap allocations. Java deserialisation involves heap manipulation that does not use regular constructors, which cannot be captured by a standard pointer analysis. The analysis compensates for this by creating pseudo-objects which have fields that are recursively set to point to serialisable objects of the specific field's declared type. Pseudo-objects are set as the arguments of a trampoline method.

2.1.3 Heap access paths. For the purposes of our dynamic analysis, we require a heap access path [23], a sequence of field dereferences from a source object to the sink object. Heap access paths can be obtained from the heap graph, constructed during the pointer analysis. A heap graph's vertices represent abstract objects, and edges represent object fields. Figure 1 shows the heap graph for the example Java code. The types `B` and `C` are subtypes of `A`. The `String` object (line 21 in Listing 1) flows from the source object

`Container` via its element field, labelled $\langle e \rangle$ in the graph and finally the field labelled by $\langle className \rangle$ in the *Element* instance. Hence, the heap access path is $Container \xrightarrow{e} Element \xrightarrow{className} String$.

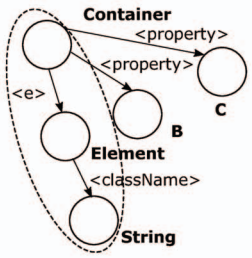

Figure 1: Heap graph.

2.2 Dynamic Analysis

During the dynamic phase, we use reflection to construct an object with properties matching the heap graph found during the static analysis phase. In our example, the heap access path, $Container \xrightarrow{e} Element \xrightarrow{className} String$, consists of `Container`, `Element` and `String` objects. After constructing the object we execute the trampoline method, which is `hashCode` in the example. Instrumentation is used to observe if we can trigger the call at the dynamic sink site.

The object graph may require further refinement for execution to reach the sink site due to guards in the code. For example, the object constructed with the heap access path will have a `Container` object with a `null` value for the `property` as that field does not appear in the heap path. Consequently, invoking the trampoline method will result in a null dereference before reaching the dynamic sink.

A technique to automatically generate random inputs that satisfy such conditions is fuzzing [27]. Generation-based or mutation-based fuzzing [26] can be used to produce random serialised data. A mutation-based fuzzer can be seeded with serialised data, which it can then mutate randomly to generate new inputs. Another approach is to use a generation-based fuzzer that generates input from a model or specification, such as the grammar for the binary serialisation format. The drawback with this approach is that we are interested in valid serialised data that must be both syntactically and semantically valid [30]. A generation-based fuzzer can produce syntactically correct data. However, it is likely that most inputs will be semantically invalid resulting in runtime exceptions. Our solution to address this is to use the heap model from the static analysis as the source for generating valid deserialisable object graphs.

In the case of the example heap graph in Figure 1, the edges that can be added to the heap access path are the two edges labelled $\langle property \rangle$ for the field property in the `Container` object. That is, the `property` field can point to either an object of type `C` or `B`. If the fuzzer generates an input with the `property` field set to type `B` it will not reach the dynamic sink. In another fuzzing run, if it picks `C`, execution will reach the dynamic sink. Once an input is generated, the trampoline method (`hashCode`) is executed with the object of type `Container` as the receiver, which is the root of the heap access path. If the sink, `Class::forName` is reached during the execution, the fuzzer has found a serialisable object that can reach a dynamic sink on deserialisation.

3 EXPERIMENTAL EVALUATION

In order to evaluate our approach, we implemented our analysis in Doop [5], a state-of-the-art datalog-based framework for specifying Java pointer-analyses. Doop was selected for its scalability in analysing large programs, its expressiveness and support for implementing custom declarative analyses.

We adapted Doop's open-program analysis features to approximate Java deserialisation as described in the analysis section. The analysis pipeline consists of points-to computation using on-the-fly call graph construction for the target Java library, applying constraints over the points-to graph to obtain flows from which heap access paths are constructed.

In our evaluation, for fuzzing we used Randoop [29], a stable and popular random testing tool [31]. It is possible that other tools would also work with this approach but we have not investigated this in our evaluation.

Randoop as a test-oriented fuzzing tool generates a sequence of statements. These method sequences are created incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences. When a new sequence is created, it is executed and checked against a set of contracts to output tests. We modified Randoop so that upon the creation of a new sequence, it selects a variable from the method sequence and passes it to a checker. The checker invokes trampoline methods on the variable and if the execution reaches a sink method, it is classified as a failing test and output on completion of test generation.

Experimental Setup. The experiments were performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 64GB of RAM on Linux Ubuntu 18.04.3. Doop[4] was run using the Java 8 platform as implemented in Oracle's version 8 of the JDK (build 1.8.0_221-b11). Doop was run with the options for a context-insensitive analysis with support for light reflection.

Dataset. We used nine libraries obtained from Frohoff's *ysoserial* repository [18]. This repository is the the most popular collection of serialisation payloads for Java binary serialisation. It is used in penetration testing tools to detect vulnerabilities and has inspired a similar effort for the .NET platform[5]. Another potential source of vulnerabilities is *marshalsec* [6] which broadens it to other serialisation mechanisms. As we specifically focus on reflection-enabled vulnerabilities, we excluded libraries with DoS vulnerabilities. We also chose libraries with vulnerabilities for `hashCode` and `compare` trampolines, since they account for nine out of 17 libraries with remote code execution vulnerabilities. For dynamic sinks, we have considered methods that load classes, and reflective calls to classes to fetch their method(s).

Availability of data and implementation. The dataset, code for the analysis and the results are available in the repository[7] for the paper.

[4]Doop version built from commit: https://bitbucket.org/yanniss/doop/commits/ba731a63c90e94f9e94afca39ff15c5082bf868d
[5]https://github.com/pwntester/ysoserial.net
[6]https://github.com/mbechler/marshalsec
[7]https://bitbucket.org/unshorn/serhybridpub/

Table 1: Overview of Experiments

Library	Time (s)	Sinks	Paths	Objects
bsh-2.0b5	655	1	1	0
clojure-1.8.0	timeout	n/a	n/a	n/a
commons-beanutils-1.9.2	786	0	0	0
commons-collections-3.1	1611	1	1	1
commons-collections4-4.0	681	1	1	1
groovy-2.3.9	5204	3	3	0
hibernate	3397	2	3	0
jython-standalone-2.5.2	timeout	n/a	n/a	n/a
rome-1.0	390	1	0	0

3.1 Preliminary Results

In evaluating our approach, the analysis reported two of the reflection-based injection attacks from the *ysoserial* collection of vulnerabilities. Table 1 shows an overview of the results from the static analysis: the running time, number of dynamic sinks detected, heap access paths from the sink to the source object with the trampoline method and finally, the number of objects created. For two libraries, *clojure* and *jython*, the points-to analysis timed out.

For the Apache Commons Collections (ACC) libraries, we were able to reproduce an object with an execution path from the trampoline to the dynamic sink with a tainted flow. We were able to produce them for two variants of payloads for the library. For the vulnerabilities that involved the `InvokerTransformer` class from the ACC library, we were able to reproduce an execution path to a dynamic sink (`getMethod`) from `hashCode` and `compare` trampolines respectively. From these, only one required further fuzzing and for the other, using reflection to just instantiate the objects on the heap access paths resulted in an object that produced the execution path on invoking the trampoline method. Randoop was able to produce an input with the aid of the heap access paths in under two minutes, and without which it produces no results within a time limit of 30 minutes. We set this limit empirically as it demonstrates the effectiveness of the static pre-analysis. In comparison, *Gadget Inspector*, the only other known analyser, produced four call chains for the ACC vulnerabilities, and in two cases, it missed the full set of gadget classes in the actual exploit. Another benefit of our analysis is that it produces executable code to produce the input that reaches the target sink, which aids in the comprehension of the analysis reports.

We also used an alternative fuzzer AFL [36] - a security-oriented greybox fuzzer that uses instrumentation for coverage guided generation of inputs. As AFL is for C programs, we used the Java front-end for AFL Kelinci [24]. In contrast to Randoop, this technique uses reflection to construct the object during fuzzing. Heap access paths are augmented with additional points-to facts from the heap graph and the trampoline method executed for vulnerabilities. Using fuzzing to produce a successful execution from the trampoline to the sink site required only under three minutes. But using fuzzing without the heap access path as a guide for the fuzzer produced no successful results after ten minutes of fuzzing.

4 RELATED WORK

Fuzzing has proven to be an effective and efficient method to discover faults in programs using random inputs. There are multiple techniques for input generation for object oriented programs: by brute-force where new objects are allocated and the field values are directly set; and constructively building inputs by using public class constructors and invoking methods on instances to change their state [7]. Korat [4] relies on the former technique, while Randoop [29] uses the latter. Finally, there are testing frameworks that utilise QuickCheck [8] style generators to randomly sample inputs. However, none of these tools are primarily for fuzzing, and input generation is usually a part of a test generation framework. Kelinci [24] fuzzes Java programs with file inputs using AFL, a security-oriented fuzzer. JFuzz [22] is a concolic execution based file input fuzzer for Java. JCrasher [9] is a largely dynamic analysis similar to Randoop. Unlike Randoop, which generates regression test suites, JCrasher tests for robustness by generating random sequences to cause undeclared runtime exceptions in a program. Check 'n' Crash [10] adds a static pre-analysis to JCrasher to produce targeted tests which are counterexamples to a specification. Most fuzzing tools have the goal of program coverage or exploring state. AFLGo [3] targets generating inputs that can reach a specific program location. In this sense, it is similar to what we attempt to achieve in our analysis with the dynamic sink as the program location that we want to reach.

A whitepaper from Hewlett Packard Enterprise [28] describes various recent serialisation-related vulnerabilities and countermeasures. Dahse et al. [15] study serialisation-based object injection attacks for PHP and build a taint analysis to generate POP gadget chains. The analysis, based on symbolically interpreting the script for tainted dataflow, is suited for the PHP scripting language. Dietrich et al. [16] study DoS attacks that use serialisation and propose an approach to mitigate the attacks.

5 CONCLUSION

We have presented a hybrid analysis to detect serialisation vulnerabilities, and we evaluated our approach on a dataset of real-world serialisation vulnerabilities. The results of our evaluation on the *ysoserial* vulnerabilities dataset, which shows two confirmed reports, are promising. The analysis uses a static pre-analysis to improve the efficiency of fuzzing. Our evaluation shows that without the results from the pre-analysis, Randoop was not able to produce a result within the time limit (30 minutes).

On the other hand, our approach can also be considered a dynamic post-analysis that improves the precision of a primarily static analysis. The only alternative method available, *Gadget Inspector*[19], may suffer precision issues due to missed branch conditions, and it requires manual inspection of the reports. The dynamic post-analysis addresses this by selecting true positives, thereby removing all false positives. This may reduce the recall of the analysis by missing true positives. However, as mentioned earlier, the analysis is already unsound like most non-trivial program analyses.

ACKNOWLEDGMENTS

We would like to thank Amjed Tahir and Lu Zhang for their valuable feedback.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks.

[2] Joshua Bloch. 2008. *Effective Java (2nd Edition) (The Java Series)* (2 ed.). Prentice Hall PTR, NJ, USA.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS âĂŹ17)*. Association for Computing Machinery, New York, NY, USA, 2329âĂŞ2344. https://doi.org/10.1145/3133956.3134020

[4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002), 123–133. http://doi.acm.org/10.1145/566171.566191

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 243–262. http://doi.acm.org/10.1145/1640089.1640108

[6] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. 2015. Understanding Caller-sensitive Method Vulnerabilities: A Class of Access Control Vulnerabilities in the Java Platform. In *Proceedings SOAP'15*. ACM, 7–12.

[7] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2007. Experimental Assessment of Random Testing for Object-oriented Software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 84–94. http://doi.acm.org/10.1145/1273463.1273476

[8] Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (May 2011), 53âĂŞ64. https://doi.org/10.1145/1988042.1988046

[9] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.* 34, 11 (Sept. 2004), 1025–1050. https://doi.org/10.1002/spe.602

[10] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'N' Crash: Combining Static Checking and Testing. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. ACM, New York, NY, USA, 422–431. https://doi.org/10.1145/1062455.1062533

[11] CVE Details 2015. CVE-2015-3253 (Vulnerability in Groovy). https://www.cvedetails.com/cve/CVE-2015-3253/. [Online; accessed 25-May-2020].

[12] CVE Details 2015. CVE-2015-4852 (Vulnerability in Oracle WebLogic Server). http://www.cvedetails.com/cve/CVE-2015-4852 [Online; accessed 25-May-2020].

[13] CVE Details 2016. CVE-2016-1000031 (Vulnerability in Struts). https://www.cvedetails.com/cve/CVE-2016-1000031. [Online; accessed 25-May-2020].

[14] CVE Details 2016. CVE-2016-4000 (Vulnerability in Jython). https://www.cvedetails.com/cve/CVE-2016-4000/. [Online; accessed 25-May-2020].

[15] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, 42–53. http://doi.acm.org/10.1145/2660267.2660363

[16] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.10

[17] Christopher Frohoff and Gabriel Lawrence. 2015. Marshalling Pickles. http://frohoff.github.io/appseccali-marshalling-pickles/ [Online; accessed 25-May-2020].

[18] Christopher Frohoff and Gabriel Lawrence. 2015. ysoserial (A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization.). https://github.com/frohoff/ysoserial [Online; accessed 25-August-2020].

[19] Ian Haken. 2018. Automated Discovery of Deserialization Gadget Chains. https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains-wp.pdf [Online; accessed 25-May-2020].

[20] Maurice P Herlihy and Barbara Liskov. 1982. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 4 (1982), 527–551.

[21] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. 2016. An In-Depth Study of More Than Ten Years of Java Exploitation. In *Proceedings CCS'16*. ACM.

[22] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A Concolic Whitebox Fuzzer for Java. In *NASA Formal Methods*.

[23] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. http://doi.acm.org/10.1145/2931098

[24] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2511–2513. http://doi.acm.org/10.1145/3133956.3138820

[25] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. http://doi.acm.org/10.1145/2644805

[26] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *CoRR* abs/1812.00140 (2018). arXiv:1812.00140 http://arxiv.org/abs/1812.00140

[27] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. http://doi.acm.org/10.1145/96267.96279

[28] A. Muñoz and C. Schneider. 2016. The Perils of Java Deserialization. https://community.hpe.com/t5/Security-Research/The-perils-of-Java-deserialization/ba-p/6838995#.WECzUsJ96cY [Online; accessed 25-May-2020].

[29] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 815–816. http://doi.acm.org/10.1145/1297846.1297902

[30] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329âĂŞ340. https://doi.org/10.1145/3293882.3330576

[31] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 201–211.

[32] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. *SIGPLAN Not.* 40, 10 (Oct. 2005), 59–76. https://doi.org/10.1145/1103845.1094817

[33] Arshan Dabirsiaghi Stefano Di Paola. 2016. Expression Language Injection. https://www.mindedsecurity.com/fileshare/ExpressionLanguageInjection.pdf. https://www.mindedsecurity.com/fileshare/ExpressionLanguageInjection.pdf [Online; accessed 25-May-2020].

[34] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham.

[35] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*.

[36] Michal Zalewski. 2017. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/technical_details.txt. http://lcamtuf.coredump.cx/afl/technical_details.txt [Online; accessed 25-May-2020].