Now You Serial, Now You Don’t — Systematically Hunting for Deserialization Exploits

Blog

Alyssa Rahman

Dec 13, 2021

17 mins read

Vulnerabilities

Threat Research

Deserialization vulnerabilities are a class of bugs that have plagued multiple languages and applications over the years. These include Exchange (CVE-2021-42321), Zoho ManageEngine (CVE-2020-10189), Jira (CVE-2020-36239), Telerik (CVE-2019-18935), Jenkins (CVE-2016-9299), and more. Fundamentally, these bugs are a result of applications placing too much trust in data that a user (or attacker) can tamper with.

Attackers have leveraged these vulnerabilities for years to upload files, access unauthorized resources, and execute malicious code on targeted servers. Within the past 2 years, Mandiant has particularly observed APT41 using .NET ViewState and Java deserialization exploits to target companies and government entities within North America.

Given the prevalence and impact of these vulnerabilities, our goal was to create a process to systematically hunt for exploitation attempts. In this blog post, we will share our new rule generation (HeySerial.py) and validation (CheckYoself.py) tools and walk through the research process we used to create them.

While this blog post mainly focuses on deserialization exploits, the tools and processes presented here can help with hunting for the exploitation of other types of zero-days. For example, we can use HeySerial to generate hunting rules for the JNDI code injection zero-day released last week for log4j (CVE-2021-44228). For more details, check the “A Note on CVE-2021-44228” section later in the post.

Understanding the Problem

What is a Deserialization Vulnerability?

“Serialized” data is just an object or data structure that has been encoded in a way that can be transferred easily – for example over the network. Developers do this regularly to pass objects between different parts of an application or between a client and server to maintain state. Once it’s transferred, it can be "deserialized" and used like it never left the original function.

Deserialization vulnerabilities result from applications putting too much trust in data that a user (or attacker) can modify. Deserialization can become dangerous when 3 conditions are met:

- The serialized object is provided by or can be modified by a user.
- An application attempts to deserialize and use the object without validation.
- The object is deserialized by a portion of the application with valuable libraries in the "class path".

Exploiting a deserialization issue involves crafting a payload that replaces what should be a benign object or data structure – such as a session token or a ViewState – with code in the targeted language that executes something malicious for the attacker.

If dangerous classes or libraries are imported and accessible in the application “class path”, an attacker can reference useful functions or object types (also referred to as “gadgets”) to execute their payload. Due to how applications are structured, the dangerous functions may not be directly accessible, so successful exploitation often requires chaining several gadgets together.

Projects such as YSoSerial (Java) and YSoSerial .NET (C#) consolidate public research on successful gadget chains for common libraries and make it easy for anyone to generate a payload with one of these chains. This is then encoded and can be passed to servers with deserialization bugs. When an application with these gadgets imported unsafely deserializes the payload, the chain will automatically be invoked and execute the embedded command on the affected server.

What Does Successful Exploitation Look Like?

Deserialization issues in HTTP servers can appear in many places – session/state data, Cookies, HTML form inputs, etc. In one recent example (CVE-2019-18211), the C1 CMS application unsafely deserialized objects passed through certain SOAP requests, leading to remote code execution.

An HTTP SOAP request with a malicious payload is shown in Figure 1, and the server response is shown in Figure 2. In this case, the server returned a simple HTTP 200 OK response after deserializing and executing the provided object.

Figure 1: CVE-2019-18211 Exploit - Request

Figure 2: CVE-2019-18211 Exploit - Response

Problem Surface Area

As mentioned, deserialization vulnerabilities can affect a wide variety of languages and libraries which results in a very large surface area for attackers. Before we can develop a comprehensive hunting strategy, we need a thorough understanding of the problem space.

Varying Protocols

This class of vulnerabilities is most commonly discussed in the context of web applications, but they’re not the only applications affected.

In 2017, Google Project Zero demonstrated how to use .NET deserialization to target systems over Managed DCOM instead of HTTP. While not specifically deserialization, in 2016, two different Black Hat talks outlined ways to exploit the Java Messaging Service (JMS) and Java Naming and Directory Interface (JNDI) APIs through object/command injection. Just last week, a JNDI command injection 0-day was released for the log4j Java logging package (CVE-2021-44228).

For this blog post, we will limit our scope to attacks that occur over HTTP. This will allow us to focus our initial research while targeting the majority of deserialization exploitation attempts.

Languages and Objects

While any language can theoretically be at risk, some of the common languages/object types exploited with this class of vulnerability are serialized Java objects, .NET ViewStates, pickled Python objects, and serialized PHP objects. The following table details the header values used as a prefix for each of these object types.

Object Type	Header (Hex)	Header (Base64)
Java Serialized	AC ED	rO
.NET ViewState	FF 01	/w
Python Pickle	80 04 95	gASV
PHP Serialized*	4F 3A	Tz

*PHP serialized objects start with the ASCII O:LENGTH_OF_NAME: where LENGTH_OF_NAME is an integer. 4F 3A is the hex encoding of O:, but further tuning with a regular expression like O:[0-9]+: is necessary to avoid false positives.

Objects may also be serialized using language-specific formatters. For example, .NET applications may use formatters such as the BinaryFormatter, LosFormatter, NetDataContractSerializer, Json.NET, or SoapFormatter depending on the vulnerable gadget. Of note, YSoSerial.NET generates UTF-16LE objects for most of its supported formatters, so many of these payloads start with 0xFFFE.

Encoding

While serialized objects *can* be transferred using the raw bytes, more frequently they will be encoded or encrypted in some way to make network transfer simpler. Base64 is very common as it is language agnostic and simple to implement.

Encoding methods can also be layered—for example by GZIP compressing and then Base64 encoding an object. For this blog post, we will focus on simple Base64 encoding due to its prevalence.

Known vs Unknown Chains

Finally, this problem space includes a lot of unknowns. Deserialization vulnerabilities are regularly disclosed and, as mentioned, can affect any part of an application. Attackers are much more limited in how they can weaponize these vulnerabilities, and as defenders, we have an opportunity to hunt for novel exploitation by looking for the gadget chains and keywords that attackers must use to execute their final payload.

Security researchers (and attackers) are constantly looking for new dangerous gadget chains in common libraries or applications. Several well-known projects that centralize this research are listed in the following table, although this is by no means a comprehensive list. For this blog post, we'll start with the ysoserial Java project, which was the first major tool and research to be released for this class of bugs.

Project Name	Language/Software Affected	GitHub URL
ysoserial	Java	https://github.com/frohoff/ysoserial
ysoserial (forked)	Java	https://github.com/wh1t3p1g/ysoserial
ysoserial.net	.NET 3.5	https://github.com/pwntester/ysoserial.net
ysoserial.net v2 branch	.NET 2 (currently only chains for v3.5)	https://github.com/pwntester/ysoserial.net/tree/v2

For this research effort we want to identify a way to generate rules for known gadget chains, but we also want a more generalized approach that can let us proactively identify exploitation attempts for novel vulnerabilities and/or gadget chains. One method here could be looking for the payload vs the chain used to execute the payload. For example, we could hunt for serialized objects with DOS headers, malicious commands, or suspicious binaries.

Solution Surface Area

Now that we understand what we are trying to hunt for, we need to determine how we will hunt for it. The available detection surface area varies depending on your goals and visibility, but hunting opportunities typically fall into three categories:

- Network
  The most direct method for detecting this method of attack is to observe the exploitation attempt as the requests are made.
- Endpoint - Dynamic
  This may involve looking for uncommon process execution or behavior from web servers. (For example, IIS servers running cmd.exe /c whoami.) This will limit us to observing successful exploitation attempts only, though, and it will be biased towards exploitation for remote code execution (RCE). We may have limited visibility into exploitation for other objectives like file upload or remote URL inclusion.

- Endpoint - Static (Log/File)
    Depending on your network traffic visibility, using YARA rules to look at decrypted requests in server logs may provide the same (or better) visibility into exploitation attempts than a network IOC. One limitation here is that logs may not include the full server response, so we will have incomplete evidence.

For this blog post, we will focus on network hunting (through Snort rules) with some static log file hunting (through YARA rules).

Make or Break

Now that we have a grasp of the problem space we want to address, we can define a hunting plan. We need an approach that will let us generate hunting logic and translate it into a variety of detection rule formats for both suspicious gadget chains and suspicious keywords. We also need to account for variability in object type and encoding method.

Tools

Since this is a complex class of bugs, we started out by creating HeySerial, a Python tool for rule generation. This lets us rapidly prototype detection logic, and it will let us keep up with new vulnerabilities as they are found.

As of publishing, HeySerial supports the following options:

Flag	Description	Options (Defaults Bolded)	Format
-k	Keyword(s)	N/A	Space delimited list of strings
-c	Gadget Chain(s)	N/A	Space delimited list of chains Chain format – <Name>::<key1>+<key2>…
-t	Object Type(s)	**JavaObj, PythonPickle, PHPObj, …**	Space delimited list of strings See help (-h) for full list.
-e	Encoding Method(s)	**base64, raw, utf8, utf16le**	Space delimited list of strings Single method and/or chain. Chain format – <method1>+<method2>
-o	Rule Output Type(s)	**snort, yara**	Space delimited list of strings
-r	Report Type(s)	**bar**, tsv	Space delimited list of strings

To generate rules for ViewState objects with a known vulnerable chain:

```
python3 heyserial.py -c 'ExampleChain::mscorlib+ActivitySurrogateSelector' -t NETViewState
```

To generate rules for all object types with suspicious keywords:

```
python3 heyserial.py -k cmd.exe whoami 'This file cannot be run in DOS mode'
```

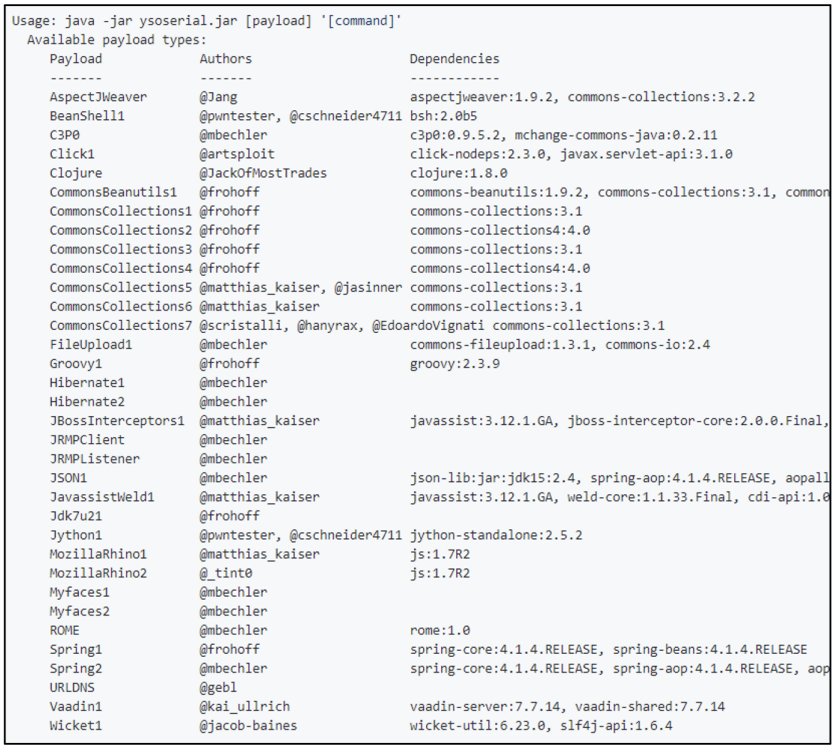
To generate rules for ViewState objects with UTF-16LE encoded Base64 encoded keywords:

```
python3 heyserial.py -k Process.Start -t NETViewState -e "base64+utf16le"
```

Although HeySerial supports a limited number of initial encoding and object types, it was designed to be extensible. For more details on how to extend HeySerial and add new encoding methods, object types, or rule formats, check out the Developer Guide.

Solve for Ex(ploits)

The next step is gathering (or creating) payloads to test our rules against. There are many public projects, as mentioned, but we will focus on the ysoserial Java gadget chains. YSoSerial lists the supported chains in the README, but the list is more than three items so let's automate it!

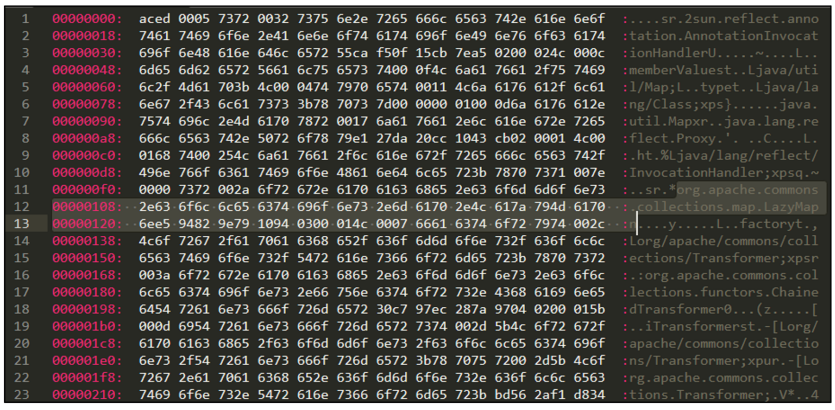


```
Usage: java -jar ysoserial.jar [payload] '[command]'
  Available payload types:
     Payload             Authors                    Dependencies
     -------             -------                    ------------
     AspectJWeaver       @Jang                      aspectjweaver:1.9.2, commons-collections:3.2.2
     BeanShell1          @pwntester, @cschneider4711 bsh:2.0b5
     C3P0                @mbechler                  c3p0:0.9.5.2, mchange-commons-java:0.2.11
     Click1              @artsploit                 click-nodeps:2.3.0, javax.servlet-api:3.1.0
     Clojure             @JackOfMostTrades          clojure:1.8.0
     CommonsBeanutils1   @frohoff                   commons-beanutils:1.9.2, commons-collections:3.1, common
     CommonsCollections1 @frohoff                   commons-collections:3.1
     CommonsCollections2 @frohoff                   commons-collections4:4.0
     CommonsCollections3 @frohoff                   commons-collections:3.1
     CommonsCollections4 @frohoff                   commons-collections4:4.0
     CommonsCollections5 @matthias_kaiser, @jasinner commons-collections:3.1
     CommonsCollections6 @matthias_kaiser           commons-collections:3.1
     CommonsCollections7 @scristalli, @hanyrax, @EdoardoVignati commons-collections:3.1
     FileUpload1         @mbechler                  commons-fileupload:1.3.1, commons-io:2.4
     Groovy1             @frohoff                   groovy:2.3.9
     Hibernate1          @mbechler
     Hibernate2          @mbechler
     JBossInterceptors1  @matthias_kaiser           javassist:3.12.1.GA, jboss-interceptor-core:2.0.0.Final,
     JRMPClient          @mbechler
     JRMPListener        @mbechler
     JSON1               @mbechler                  json-lib:jar:jdk15:2.4, spring-aop:4.1.4.RELEASE, aopall
     JavassistWeld1      @matthias_kaiser           javassist:3.12.1.GA, weld-core:1.1.33.Final, cdi-api:1.0
     Jdk7u21             @frohoff
     Jython1             @pwntester, @cschneider4711 jython-standalone:2.5.2
     MozillaRhino1       @matthias_kaiser           js:1.7R2
     MozillaRhino2       @_tint0                    js:1.7R2
     Myfaces1            @mbechler
     Myfaces2            @mbechler
     ROME                @mbechler                  rome:1.0
     Spring1             @frohoff                   spring-core:4.1.4.RELEASE, spring-beans:4.1.4.RELEASE
     Spring2             @mbechler                  spring-core:4.1.4.RELEASE, spring-aop:4.1.4.RELEASE, aop
     URLDNS              @gebl
     Vaadin1             @kai_ullrich               vaadin-server:7.7.14, vaadin-shared:7.7.14
     Wicket1             @jacob-baines              wicket-util:6.23.0, slf4j-api:1.6.4
```

Figure 3: YSoSerial Java payload options

The following command generates a payload that launches “calc.exe” using the CommonsCollections1 chain. The raw hex bytes of the payload are shown in Figure 4. If we print the file contents directly, we will see some interesting strings mixed in with other non-printable (non-ASCII) characters, as seen in Figure 5.

```
java -jar utils/ysoserial.jar CommonsCollections1 calc.exe > commonscollections1.bin
```



```
00000000: aced 0005 7372 0032 7375 6e2e 7265 666c 6563 742e 616e 6e6f  ....sr.2sun.reflect.anno
00000018: 7461 7469 6f6e 2e41 6e6e 6f74 6174 696f 6e49 6e76 6f63 6174  tation.AnnotationInvocat
00000030: 696f 6e48 616e 646c 6572 55ca f50f 15cb 7ea5 0200 024c 000c  ionHandlerU.....~....L..
00000048: 6d65 6d62 6572 5661 6c75 6573 7400 0f4c 6a61 7661 2f75 7469  memberValuest..Ljava/uti
00000060: 6c2f 4d61 703b 4c00 0474 7970 6574 0011 4c6a 6176 612f 6c61  l/Map;L..typet..Ljava/la
00000078: 6e67 2f43 6c61 7373 3b78 7073 7d00 0000 0100 0d6a 6176 612e  ng/Class;xps}.......java.
00000090: 7574 696c 2e4d 6170 7872 0017 6a61 7661 2e6c 616e 672e 7265  util.Mapxr..java.lang.re
000000a8: 666c 6563 742e 5072 6f78 79e1 27da 20cc 1043 cb02 0001 4c00  flect.Proxy.'. ..C....L.
000000c0: 0168 7400 254c 6a61 7661 2f6c 616e 672f 7265 666c 6563 742f  .ht.%Ljava/lang/reflect/
000000d8: 496e 766f 6361 7469 6f6e 4861 6e64 6c65 723b 7870 7371 007e  InvocationHandler;xpsq.~
000000f0: 0000 7372 002a 6f72 672e 6170 6163 6865 2e63 6f6d 6d6f 6e73  ..sr.*org.apache.commons
00000108: 2e63 6f6c 6c65 6374 696f 6e73 2e6d 6170 2e4c 617a 794d 6170  .collections.map.LazyMap
00000120: 6ee5 9482 9e79 1094 0300 014c 0007 6661 6374 6f72 7974 002c  n....y.....L..factoryt.,
00000138: 4c6f 7267 2f61 7061 6368 652f 636f 6d6d 6f6e 732f 636f 6c6c  Lorg/apache/commons/coll
00000150: 6563 7469 6f6e 732f 5472 616e 7366 6f72 6d65 723b 7870 7372  ections/Transformer;xpsr
00000168: 003a 6f72 672e 6170 6163 6865 2e63 6f6d 6d6f 6e73 2e63 6f6c  .:org.apache.commons.col
00000180: 6c65 6374 696f 6e73 2e66 756e 6374 6f72 732e 4368 6169 6e65  lections.functors.Chaine
00000198: 6454 7261 6e73 666f 726d 6572 30c7 97ec 287a 9704 0200 015b  dTransformer0...(z.....[
000001b0: 000d 6954 7261 6e73 666f 726d 6572 7374 002d 5b4c 6f72 672f  ..iTransformerst.-[Lorg/
000001c8: 6170 6163 6865 2f63 6f6d 6d6f 6e73 2f63 6f6c 6c65 6374 696f  apache/commons/collectio
000001e0: 6e73 2f54 7261 6e73 666f 726d 6572 3b78 7075 7200 2d5b 4c6f  ns/Transformer;xpur.-[Lo
000001f8: 7267 2e61 7061 6368 652e 636f 6d6d 6f6e 732e 636f 6c6c 6563  rg.apache.commons.collec
00000210: 7469 6f6e 732e 5472 616e 7366 6f72 6d65 723b bd56 2af1 d834  tions.Transformer;.V*..4
```

Figure 4: CommonsCollections1 payload

Figure 5: Inspecting a CommonsCollections1 YSoSerial payload

We can use a simple Bash script to loop through the list of supported chains and output them to a file. Some chains require different payload inputs than others, so check out generate_payloads.sh to see how we can handle that with a switch-case block. For YSoSerial.NET payloads, try the generate_payloads.ps1 script on a Windows system.

Since we want to test Base64 rules, we can also add a command to the payload generation script that will save the Base64 encoded payloads to a separate file for us.

```
base64 -w 0 “$filename.bin” > “$filename.base64”
```

PCAPs or It Didn’t Happen

Now that we have our generated payloads, we can simply run YARA against the files to validate our rules. However, to test our Snort rules, we’ll need to generate a PCAP with the payloads demonstrating a “malicious” web request.

For that we can start a basic HTTP server using this server.py script, and we can execute the following Bash commands in a separate terminal. This will do a plain HTTP POST request to our local server with each payload in the body of the request.

```
for sample in `ls *.bin`; do curl -m 3 --data-binary "@$sample" http://127.0.0.1:12345; done;
```

```
for sample in `ls *.base64`; do curl -m 3 --data "@$sample" http://127.0.0.1:12345; done;
```

For testing purposes, we don't need to exploit an actual vulnerability or web server. We just need network traffic that looks like an attempted exploit. Using these PCAPs and the payload files we already generated, we will be able to test both Snort and YARA rules.

The Golden Rules

Ok we have payloads, and we have a script. It's finally time to make our hunting rules! There are many ways to identify the list of keywords we want to include for each chain. The YSoSerial repo includes details of the classes and functions that make up a payload, but this is

1. Scattered in different files
2. Not necessarily an exact match for what will be in the final serialized object

Another simple method for bulk extracting imported classes or functions is looping through all (raw hex) payloads and getting the first 5 strings with a "\..*\." pattern. We can accomplish this by adding the following line to our Bash script:

```
strings "$filename.bin" | grep -E '\..*\.' | head -5 > "$filename.strings"
```

This approach requires a bit of manual cleanup on the strings, but once we're done, we can generate rules knowing these classes or functions are included in a default YSoSerial Java payload.

(*Please note, these—especially YARA—are all hunting rules for research purposes, not detections.* ***Do not*** *deploy these to production systems without testing and tuning for your environment.*)

The following command generates Snort and YARA rules with both raw and Base64 encoded chains for some common payloads in YSoSerial (Java).

```
python3 heyserial.py -t JavaObj -e base64 raw -o snort yara -c "AspectJWeaver::HashSet+TiedMapEntry+org.apache.com
"BeanShell1::java.util.PriorityQueue+Comparator+java.lang.reflect.Proxy+Hashtable+Vector"
"C3P0::com.mchange.v2.c3p0.PoolBackedDataSource+AbstractPoolBackedDataSource+PoolBackedDataSourceBase+com.mchange.v
"Click1::java.util.PriorityQueue+org.apache.click.control.Column+Column+Table+AbstractControl" "Clojure::HashMap+cl
"CommonsBeanutils1::java.util.PriorityQueue+org.apache.commons.beanutils+BeanComparator+ComparableComparator+com.su
"CommonsCollections1 3::sun.reflect.annotation.AnnotationInvocationHandler+Map+Proxy+org.apache.commons.collections
"CommonsCollections2::java.util.PriorityQueue+org.apache.commons.collections4.comparators.TransformingComparator+Co
"CommonsCollections4::java.util.PriorityQueue+org.apache.commons.collections4.comparators.TransformingComparator+Co
"CommonsCollections5::javax.management.BadAttributeValueExpException+org.apache.commons.collections.keyvalue.TiedMa
"CommonsCollections6::java.util.HashSet+org.apache.commons.collections.keyvalue.TiedMapEntry+org.apache.commons.col
"CommonsCollections7::java.util.Hashtable+org.apache.commons.collections.map.LazyMap+ChainedTransformer+ConstantTra
"Groovy1::sun.reflect.annotation.AnnotationInvocationHandler+Map+Proxy+org.codehaus.groovy.runtime.ConvertedClosure
"Hibernate1 2::java.util.HashMap+org.hibernate.engine.spi.TypedValue+org.hibernate.type+ComponentType+AbstractType+
"JavassistWeld1::org.jboss.weld.interceptor.proxy.InterceptorMethodHandler+org.jboss.weld.interceptor.builder.Inter
"JBossInterceptors1::org.jboss.interceptor.proxy.InterceptorMethodHandler+org.jboss.interceptor.builder.Interceptio
"Jdk7u21::java.util.LinkedHashSet+HashSet+com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl+javax.xml.tran
"JRMPClient::java.rmi.registry.Registry+java.lang.reflect.Proxy+java.rmi.server.RemoteObjectInvocationHandler"
"JRMPListener::sun.rmi.server.ActivationGroupImpl+java.rmi.activation.ActivationGroup+java.rmi.server.UnicastRemote
"Jython1::java.util.PriorityQueue+java.util.Comparator+java.lang.reflect.Proxy+org.python.core.PyFunction+org.pytho
"MozillaRhino1::org.mozilla.javascript.NativeError+org.mozilla.javascript.NativeJavaObject+org.mozilla.javascript.M
"MozillaRhino2::org.mozilla.javascript.NativeJavaObject+org.mozilla.javascript.tools.shell.Environment+org.mozilla.
"Myfaces1 2::java.util.HashMap+org.apache.myfaces.view.facelets.el.ValueExpressionMethodExpression+javax.el.MethodE
"ROME::java.util.HashMap+com.sun.syndication.feed.impl.ObjectBean+com.sun.syndication.feed.impl.CloneableBean+java.
"Spring1 2::org.springframework.core.SerializableTypeWrapper+MethodInvokeTypeProvider+TypeProvider+java.lang.reflec
"Vaadin1::javax.management.BadAttributeValueExpException+com.vaadin.data.util.PropertysetItem+com.vaadin.data.util.
"Wicket1::org.apache.wicket.util.upload.DiskFileItem+java.io.File"
```

Putting It to the Test

Now for the exciting part—seeing it all in action!

YARA Rules

Testing YARA rules on files is really simple with YARA installed. We can save our generated rules to a file, and then run the following command:

```
yara ysoserial_CommonsCollections1.yar ysoserial_CommonsCollection1.bin
```

If it works, we will see a line with the rule name and the file it matched on. The very last line of Figure 6 shows that our YARA rule matches!

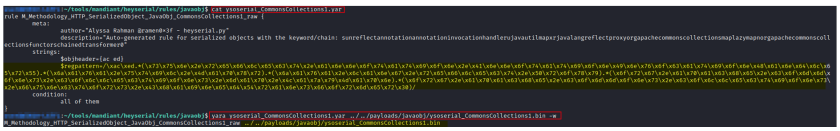


Figure 6: Testing a YARA rule

Snort Rules

We can also test out our Snort rules using a local installation of Snort. Snort will throw an error if you have invalid (or duplicate) signature IDs, so we can bulk edit our rules with this command. (*Please* update the SIDs to valid values if you deploy them in your environment.)

```
perl -pe 'BEGIN{$A=100;} s/<REPLACE_SID>/$A++/ge' -i rules/*/*snort
```

The following command will run a specific rule file against a test PCAP:

```
sudo snort -A console -k none -q -r ysoserial_java_rawbase64.pcap -c CommonsCollections1_3.snort
```

The "-k none" option tells Snort to disable checksum mode, because our test data was generated with localhost as the source/destination and will be ignored by Snort otherwise. We can also set this by changing "config checksum_mode: all" to "none" in the /etc/snort/snort.conf configuration file. As shown in Figure 7, our Snort rule matches on the PCAP!

Figure 7: Testing a Snort rule

Another Tool??

We *can* validate our rules manually, but that doesn't serve our goal of rapidly prototyping a bunch of hunting and detection ideas. To that end, we made CheckYoself, a Python script that accepts file or directory paths to Snort and YARA rules and runs it against specified data files. By default it prints a TSV of the results to the screen, but we can save this to a file as well.

This command will run all generated JavaObj rules against our JavaObj payloads and PCAPs.

```
python3 utils/checkyoself.py -y rules/javaobj -s rules/javaobj -d payloads/javaobj pcaps/ -o java_all
```

Adding the --misses flag will filter our results to show only the files (for YARA) or rules (for Snort) that had 0 matches.

```
python3 utils/checkyoself.py -y rules/javaobj -s rules/javaobj -d payloads/javaobj pcaps/ -o java_all --misses
```

What Else?

The research process and the tool we've discussed today are a great starting point, but as with any detection there are both limitations and opportunities to be aware of.

Fine Tuning Our Snort Rules

One issue with our Snort rules is that, in their current state, we have no way of knowing if the exploit attempt was successful or not which means we could get a lot of noisy alerts for internet facing systems that get scanned.

To account for this, we can use Snort flowbits. By adding flowbits:set,heyserial; to our Snort rules, we can deploy another rule like the following that looks for server responses. This example is looking for any HTTP traffic with the flowbit set from our previous HeySerial generated rules *except* for HTTP responses with a 301 (redirect) or 404 (not found) status code.

```
alert tcp any any -> any any ( msg:"M.Methodology.HTTP.SerializedObject.[ServerResponse]"; content:"HTTP";
depth:4; content:!"301"; offset:9; depth:3; content:!"404"; offset:9; depth:3;  flowbits:isset,heyserial;
threshold:type limit,track by_src,count 1,seconds 1800; sid: <REPLACE_SID>; rev:1; )
```

For production deployment, this will require additional testing and tuning to get higher fidelity results. Once we have more strict conditions around what we consider successful exploitation (or close enough that we want to review it), we will only need to monitor the ServerResponse rule.

Some examples of tuning opportunities to explore include:

- Ignoring certain HTTP status codes
- Filtering out known default pages (such as your company home page)
- Creating separate flowbits per language. This could allow us to verify that server responses match the requests. For example, a Java exploit chain request with a response from an IIS server is unlikely to be successful exploitation.

Encryption

We have tested these rules on unencrypted objects so far, but there are two cases where encryption may come into play and reduce the efficacy.

First, some objects may be encrypted. Multiple widely exploited CVEs resulted from applications encrypting .NET ViewState objects either using a static/default encryption key or by allowing users to brute force the encryption key. Attackers can use public tools to encrypt their payloads with known keys if they're able to discover or brute force one.

Second, the network traffic itself will likely be encrypted. If your network appliances are not intercepting and decrypting traffic, then you'll have to rely on other static or endpoint detections.

Opportunities are Endless!

HeySerial currently only supports network rules for HTTP traffic, but this isn't the only protocol that can be affected. Expanding these hunting rules to other protocols such as COM is likely a fruitful area for further research.

Finally some language specific formatters will still leave strings that can be detected with Hex or Base64 rules, but some may require customized encoders.

A Note on CVE-2021-44228

On December 9, 2021, a zero-day exploit was released for log4j, a Java log library. While this is an example of JNDI code injection, not necessarily deserialization, this tool and the concepts we've discussed also apply here. By adding a JNDI object prefix of "${jndi:", we can generate hunting rules for this type of command injection using the following HeySerial command.

```
python3 heyserial.py -t JNDIObj -e raw base64 -k dns:/ ldap:/ ldaps:/ rmi:/
```

These rules look for any objects that follow the unobfuscated format: ${jndi:ldap://<example>.com/a}. However, due to the ease of obfuscating this initial stage of exploitation, it will likely be more robust to focus detection on the stage two and post-exploitation stages of these attacks—such as the remote loading of serialized Java classes.

Hunting rules, sample payloads, and a test PCAP for the unobfuscated POC are provided in the HeySerial repository for your testing. Please note, these are not production/blocking ready detections.

Conclusion

In this blog post, we explored deserialization vulnerabilities and developed a process and tools to rapidly prototype detections for in-the-wild exploitation. Although this type of bug has been around for years, Mandiant continues to observe threat actors, including advanced groups like APT41, using publicly disclosed exploit "chains" in their intrusions.

Our tool, HeySerial.py, is intended to be an extensible framework that can be expanded to support additional object types, encoding methods, and rule formats. To find out more, check out our Developer Guide.

Mandiant Security Validation Content

Mandiant Security Validation includes Actions for the YSoSerial Java payloads shared in the HeySerial repository. Please see actions with VID A102-150 through A102-205 in Mandiant Advantage for more details.

Acknowledgements

Special thanks to James Hovious for sharing his expertise (and exploits), Ashley Zaya for her review, and to Gregory LeBlanc and William Ballenthin for code review. *Extra* special thanks to Evan Reese for being a Snort guru, because otherwise this blog post would probably only include YARA.

Prior Work / Additional Resources

Tools

- Deserialization-Cheat-Sheet – @GrrrDog
- Ysoserial - @frohoff
- Ysoserial (forked) - @wh1t3p1g
- Ysoserial.NET and v2 branch- @pwntester
- ViewGen – 0xacb
- Rogue-JNDI – @veracode-research

Vulnerabilities

- Log4J (CVE-2021-44228)
- Exchange (CVE-2021-42321)
- Zoho ManageEngine (CVE-2020-10189)
- Jira (CVE-2020-36239)
- Telerik (CVE-2019-18935)

- C1 CMS (CVE-2019-18211)
- Jenkins (CVE-2016-9299)
- What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability. – @breenmachine, FoxGloveSecurity (2015)

Talks and Write-Ups