Introduction -

Introduction to Serialization

Introduction

`Serialization` is the process of taking an object from memory and converting it into a series of bytes so that it can be stored or transmitted over a network and then reconstructed later on, perhaps by a different program or in a different machine environment.

`Deserialization` is the reverse action: taking serialized data and reconstructing the original object in memory.

Many object-oriented programming languages support serialization natively, including, but not limited to:

- Java
- Ruby
- Python
- PHP
- C#

For the duration of this module, we will only focus on `Python` and `PHP`; however, please note that the same concepts taught may be reapplied to most, if not all, languages that support serialization.

PHP Serialization

As an example, this is how we would `serialize an array in PHP`:

mayala@htb[/htb] `$ php -a Interactive shell php > $original_data = array("HTB", 123, 7.77); php > $serialized_data = serialize($original_data); php > echo $serialized_data; a:3:{i:0;s:3:"HTB";i:1;i:123;i:2;d:7.77;} php > $reconstructed_data = unserialize($serialized_data); php > var_dump($reconstructed_data); array(3) { [0]=> string(3) "HTB" [1]=> int(123) [2]=> float(7.77) }`

As you can see, `$original_data` is an array containing one `string` (`"HTB"`), one `integer` (`123`), and one `double` (`7.77`). Using the `serialize` function, the array is

turned into bytes that represent the array. We carry on to `unserialize` this serialized string and restore the original array as verified by the `var_dump` of `$reconstructed_data`.

Serialized objects in PHP are easy to read, unlike serialized objects in many other languages, which may look like complete gibberish to the human eye, as you will see in the Python example, but before that, let's understand what the letters and numbers in the serialized data mean:

Code: php

```
a:3:{ // (A)rray with (3) items
    i:0;s:3: "HTB"; // (I)ndex (0); (S)tring with length (3) and value:
"HTB"
    i:1;i:123; // (I)ndex (1); (I)nteger with value (123)
    i:2;d:7.77; // (I)ndex (2); (D)ouble with value (7.77)
}
```

Python Serialization

Similar to the PHP example above, we will `serialize an array in Python`. There are multiple libraries for Python which implement serialization, such as PyYAML and JSONpickle. However, Pickle is the native implementation, and it is what will be used in this example.

mayala@htb[/htb] `$ python3 Python 3.10.7 (main, Sep 8 2022, 14:34:29) [GCC 12.2.0] on linux Type "help", "copyright", "credits" or "license" for more information. >>> import pickle >>> original_data = ["HTB", 123, 7.77] >>> serialized_data = pickle.dumps(original_data) >>> print(serialized_data) b'\x80\x04\x95\x16\x00\x00\x00\x00\x00\x00\x00]\x94(\x8c\x03HTB\x94K{G@\x1f\x14z \xe1G\xae\x14e.' >>> reconstructed_data = pickle.loads(serialized_data) >>> print(reconstructed_data) ['HTB', 123, 7.77]`

Reading the serialized data `pickle` outputs is much harder than reading the output PHP provides. However, it is still possible. According to comments in the `pickle` library, a `pickle` is a `program` for a `virtual pickle machine` (PM). The `PM` contains a `stack` and a `memo` (long-term memory), and a `pickled` object is just a sequence of `opcodes` for the `PM` to execute, which will recreate an arbitrary object on the `stack`.

The PM's `stack` is a Last-In-First-Out (LIFO) data structure. You may `push` items onto the `top` of the stack, and you may `pop` the `top` object `off` of the stack.

Quoting from comments in the `pickle` library, the PM's `memo` is a "data structure which remembers which objects the pickler has already seen, so that shared or recursive objects are

pickled by reference and not by value."

In Lib/pickle.py (Python 3.10), we can see all of the `pickle opcodes` defined, and by referring to them, as well as the source code for the various pickling functions, we can piece together what our `serialized_data` does exactly when it is passed to `pickle.loads()`:

Code: python

```
'\x80\x04'
# PROTO 4
# Tell the PM that we are using protocol version 4. This is the default
since Python 3.8.
# Protocol versions 3-5 can not be unpickled by Python 2.x.

'\x95\x16\x00\x00\x00\x00\x00\x00\x00'
# FRAME 16
# Essentially we are telling the PM that the serialized data is 16 bytes
long.
# The argument is calculated like this:
# `struct.pack("<Q",
len(b']\x94(\x8c\x03HTB\x94K{G@\x1f\x14z\xe1G\xae\x14e.')) =
b'\x16\x00\x00\x00\x00\x00\x00\x00'`.

']'
# EMPTY_LIST
# Pushes an empty list onto the stack.
# Eventually, we will append the items to this list after we have defined
them.

'\x94'
# MEMOIZE
# This stores the object on the top of the stack in the 'memo' which is akin
to long-term memory.
# The memo is used to keep transient objects alive during pickling.
# In this case we are 'memozing' the empty list we just pushed onto the
stack.
# This opcode is called when pickling any of the following types:
# - __reduce__
# - bytes
# - bytearray
# - string
# - tuple
# - list
# - dict
# - set
# - frozenset
```

```
# - global

'('
# MARK
# Pushes the special 'markobject' on the stack.
# This will be referred to later as the starting point for our array items.

'\x8c\x03HTB'
# SHORT_BINUNICODE 3 HTB
# Pushes the unicode string with length 3 'HTB' onto the stack.

'\x94'
# MEMOIZE
# We tell the PM to 'memoize' the string that we just pushed onto the stack.

'K{'
# BININT1 {
# Pushes a 1-byte unsigned int with value 123 onto the stack.
# '{' is the byte representation of 123 calculated as so:
# `chr(123) = b'{'`

'G@\x1f\x14z\xe1G\xae\x14'
# BINFLOAT @\x1f\x14z\xe1G\xae\x14
# Pushes a float with the value 7.77 onto the stack.
# '@\x1f\x14z\xe1G\xae\x14' is the 8-byte float encoding of 7.77 which is
calculated like this:
# `struct.pack(">d", 7.77) = b'@\x1f\x14z\xe1G\xae\x14'`

'e'
# APPENDS
# We are telling the PM to extend the empty list on the stack with all items
we just defined back up until the 'markobject' we defined earlier.

'.'
# STOP
# This is how we tell the PM we are at the end of the pickle.
# The original array `['HTB', 123, 7.77]` was recreated and now sits at the
top of the stack.
```

Introduction to Deserialization Attacks

Introduction

As was stated in the previous section, `deserialization` is the reverse action to `serialization`, specifically taking in serialized data and reconstructing the original object in memory.

If an application ever deserializes `user-controlled` data, then there is a possibility for a `deserialization attack` to occur. An attack would involve taking serialized data generated by the application and modifying it for our benefit or perhaps generating and supplying our own serialized data.

History

Deserialization has been known as an attack vector since 2011, but it only went viral in 2016 with the `Java Deserialization Apocalypse`. This was the result of a talk delivered in 2015, in which security researchers @frohoff and @gebl explained deserialization attacks in great detail and released the infamous tool for generating Java deserialization payloads named ysoserial.

Nowadays, insecure deserialization features in the OWASP Top 10 under the `A08:2021-Software and Data Integrity Failures` category and many CVEs are published each year regarding this topic.

Attacks

Throughout this module, we will cover two primary `deserialization attacks`:

- Object Injection
- Remote Code Execution

`Object Injection` means modifying the serialized data so that the server will receive unintended information upon deserialization. For example, imagine a serialized object containing a user's role on the website. If we had control of this object, we could modify it so that when the server deserializes the object, it will instead say we have an administrative role.

`Remote Code Execution` is self-explanatory: in this attack, we supply a serialized payload which results in command execution upon being deserialized on the server side.

Identifying Serialization

White-Box

When we have access to the source code, we want to look for specific function calls to identify possible deserialization vulnerabilities quickly. These functions include (but are certainly not limited to):

- `unserialize()` - PHP
- `pickle.loads()` - Python Pickle
- `jsonpickle.decode()` - Python JSONPickle
- `yaml.load()` - Python PyYAML / ruamel.yaml
- `readObject()` - Java
- `Deserialize()` - C# / .NET
- `Marshal.load()` - Ruby

Black-Box

If we do not have access to the source code, it is still easy to identify serialized data due to the distinct characteristics in serialized data:

- If it looks like: `a:4:{i:0;s:4:"Test";i:1;s:4:"Data";i:2;a:1:{i:0;i:4;}i:3;s:7:"ACADEMY";}` - PHP
- If it looks like: `(lp0\nS'Test'\np1\naS'Data'\np2\na(lp3\nI4\naaS'ACADEMY'\np4\na.` - Pickle Protocol 0, default for Python 2.x
- Bytes starting with `80 01` (Hex) and ending with `.` - Pickle Protocol 1, Python 2.x
- Bytes starting with `80 02` (Hex) and ending with `.` - Pickle Protocol 2, Python 2.3+
- Bytes starting with `80 03` (Hex) and ending with `.` - Pickle Protocol 3, default for Python 3.0-3.7
- Bytes starting with `80 04 95` (Hex) and ending with `.` - Pickle Protocol 4, default for Python 3.8+
- Bytes starting with `80 05 95` (Hex) and ending with `.` - Pickle Protocol 5, Python 3.x
- `["Test", "Data", [4], "ACADEMY"]` - JSONPickle, Python 2.7 / 3.6+
- `- Test\n- Data\n- - 4\n- ACADEMY\n` - PyYAML / ruamel.yaml, Python 3.6+
- Bytes starting with `AC ED 00 05 73 72` (Hex) or `rO0ABXNy` (Base64) - Java
- Bytes starting with `00 01 00 00 00 ff ff ff ff` (Hex) or `AAEAAAD/////` (Base64) - C# / .NET
- Bytes starting with `04 08` (Hex) - Ruby

Some tools have been developed to detect serialized data automatically. For example Freddy is an extension for BurpSuite which aids with the detection and exploitation of `Java/.NET` serialization.

Onwards

Now that we've covered serialization and deserialization attacks at a high level let's dive deep into exploiting both PHP and Python deserialization vulnerabilities.