Chapter 14
Serialization

Revision 3.2, December 2025.

Objectives

- Review the purpose of serialization and how it works.
- Understand the potential security problems associated with serialization.
- Understand the multi-layer approach to remediating serialization attacks.

14.1 The Basics of Serialization

Programmers routinely work with data objects in memory, but sometimes the objects need to be sent over a network or written to persistent storage (typically a file) to save part of the state of the program. Serialization is a technique that allows you to package your data objects in a consistent form for storage or transmission, and then later restored to their in-memory form, either on the original machine or a different one.

If your data consists of simply types, such as numbers or strings, you could just directly output the bytes of data and then important them again later. However, there are a couple of things that make this strategy ineffective in the general case.

Complex representations: If your object is part of a more complex data structure, such as a tree. It is likely to contain pointers that form the links between the nodes in the tree. These pointers are memory addresses in the address space of the program that created the object. When the object is loaded back into memory of a possibly different program, the object is likely to reside at a new memory location so the addresses will be incorrect. You would have to understand the structure of the object that was saved to know how to adjust these addresses when loading it again.

Multiple architectures: Data representations may vary from architecture to architecture. For example, integers on one system might be represented in big endian form and on another system in little endian form[1]. In fact, some processors – such as the ARM, MIPS, and IBM PowerPC – can switch between these two modes. If you saved an object that contained a number

[1] A big-endian system stores the most significant byte of a number at the smallest memory address and the least significant byte at the largest. A little-endian system, in contrast, stores the least-significant byte at the smallest address.

from a system with a big endian processor and loaded it into one with a little endian processor, the number would have the wrong value. You would have to know that the specific bytes of the object contained a number and to know whether to rearrange the bytes.

> Serialization provides a standard byte representation of the value of data so that it can be corrected saved and restored or sent over a network, even when the computers may be of different architecture and software.

Serialization fundamentally works in similar ways across most languages and implementations, although the specifics vary greatly depending on the style and nuances of the particular language. Class implementers can explicitly declare when objects should be serializable, in which case a standard library handles everything, and this works fine but only for objects that contain simple values. Objects with complex internal state often need a custom implementation for serialization, typically by overriding a method of the standard library. The trick is understanding what the standard implementation does, its limitations, and when and how to handle serialization when appropriate.

> The easiest security mistake to make with serialization is to inappropriately trust the provider of the serialized data.

The act of deserialization converts the data from the format in which it was stored to the internal representation used by your programming language, with few if any checks as to whether the encoded data was corrupted or intentionally designed to be malicious. This is a dangerous assumption and the basis on which attacks are made. When custom code handles serialization, it needs to avoid all of the usual security pitfalls while reconstructing properly initialized objects. Objects that contain or reference other objects need special care in determining which of the objects need to also be serialized and understanding how those objects in turn work under serialization. When the source of serialized data is potentially untrustworthy, there is usually no way to defensively check for validity.

Serialization is a valuable and safe mechanism when you have full control of the data you receive for deserialization. There are a couple of general scenarios where serialization makes sense. The first scenario is when you want to save a complex object for later use. Once you produce the serialized version of the object, you can write it safely to a file, making sure that the protections are set correctly to prevent possible tampering. At some later time, your program could read the object and deserialize it, knowing that it originated from a safe source.

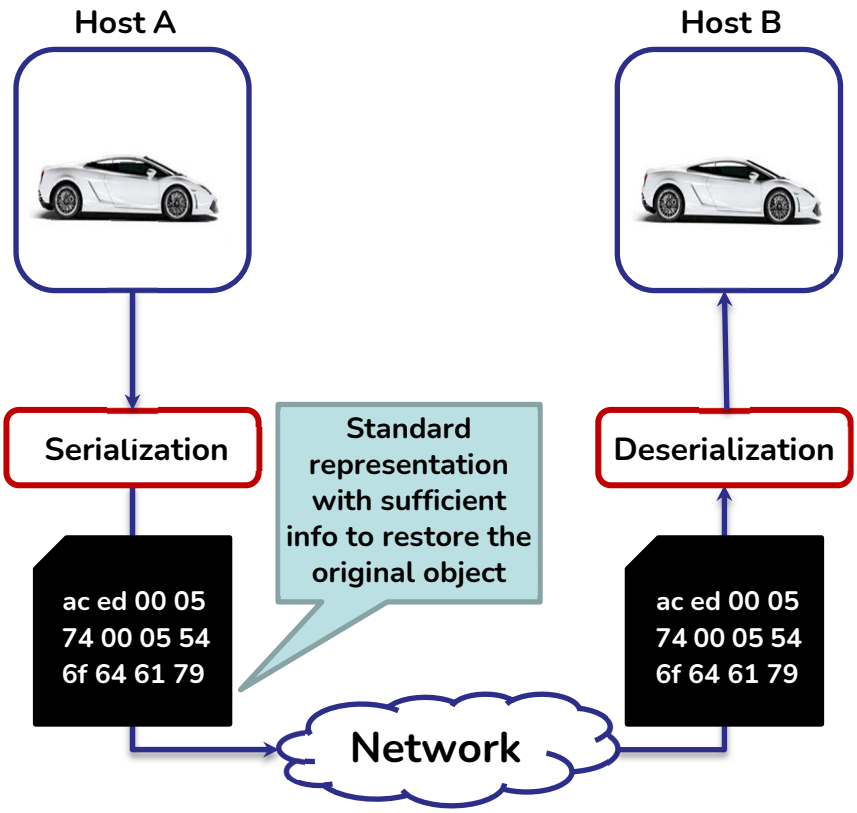

Figure 1: Conceptual Diagram of the Flow of Serialized Data

A second scenario is where you want to send a complex object from one protected server to another. In this case, you control both the sender (the program doing the serialization) and the receiver (the program doing the deserialization). Of course, you need to make sure that you send the serialized data over an encrypted tamper-proof channel, using a secure protocol such as TLS.

Attempting to deserialize any data other than valid serialized data is dangerous. This warning applies to any data an attacker might be able to modify. Deserializing damaged or maliciously modified data will generally result in indeterminate behavior, and that is a ripe opportunity for attackers to craft attacks.

While the formats used to represent serialized data will vary from language to language, at a high level, their structure and form is similar. Serialized data usually begins with a header denoting what it is – to help reject random data that might be mistakenly used – and often contains a version number

allowing future implementation changes while maintaining backward compatibility.

The serialization process starts by generating header information, that is metadata that specifies the class of the data and the format in which it will be encoded. Then, the actual data values are emitted in a predetermined sequence and, for complex objects, serialization proceeds recursively over the contained objects.

Serialization is an abstract concept potentially applicable to all kinds of software objects, so let's look at a concrete example in Java. Host A has an object that it wants to communicate to a different Host B (that may be a completely different implementation) over a common network. Using Java's standard serialization library, a sequence of bytes is generated that contains sufficient information to describe the object's metadata and values. These bytes are then sent to its peer that uses the complementary deserialization library to decode the data, create an object of the correct type, and then initialize the object to have an identical value to the original.

Serialization allows an object to be sent across implementation boundaries via a standardized byte representation. However, for everything to work safely, the data must be protected against leakage or tampering to be secure.

14.2 Serialization in Various Languages

Before considering the security aspects of serialization, we provide a brief overview of how serialization works in several popular languages and environments. This information is summarized in Figure 1 and explained below.

Python uses the standard `pickle` library to handle serialization: `dumps` to serialize and `loads` to deserialize. Note that in Section 14.5, we look at the details of a way to attack Python serialization.

Ruby serialization is handled by the `Marshal` module in a similar way to Python.

Serialization for the C++ Boost classes is based on the `iostream` `>>` and `<<` operators[2]. Basically, you can send a serialized form of the object to an iostream with `>>` and recover it with `<<`.

For the Microsoft Foundation Class (MFC) Library in C++ Visual Studio[3], serialization and deserialization is implemented by overriding the

[2] Note that the `&` operator can also be used in a similar way.

[3] `https://learn.microsoft.com/en-us/cpp/mfc/serialization-in-mfc?view=msvc-170`

`Serialize` method. Serialized objects are put in a `CArchive` object and then written to a file using the `CFile` class. This process is reversed for deserialization. The `Serialize` method does both serialization and deserialization, depending on how the member function `IsStoring` is set.

Java	
Serialize	Method: `writeObject` Implemented in: `ObjectOutputStream`
Deserialize	Method: `readObject` Implemented in: `ObjectInputStream`
Python	
Serialize	`pickle.dumps(...)` Customize pickling by defining your own `__reduce__` method.
Deserialize	`pickle.loads(...)`
Ruby	
Serialize	`Marshal.dump(...)`
Deserialize	`Marshal.load(...)`
C++ using Boost	

Serialize	`boost::archive::text_oarchive oa (filename);`
	`oa << data;`
	Invokes `serialize` method. To serialize objects in a user-defined class, you define a `serialize` method in that class.

Deserialize	`boost::archive::text_iarchive ia (filename);`
	`ia >> newdata;`
	Invokes `serialize` method. User-defined classes handled in the same way as the serialization case.

MFC – Microsoft Foundation Class Library	
Both	Derive your Class from `CObject`. Override the `Serialize` member function. `IsStoring` indicates if `Serialize` is storing or loading data.

Figure 2: Summary of Serialization Mechanisms for Several Languages and Environments

14.3 How Serialization Works

We will start by looking at how Java serialization works on an object containing four integers with values `{1,2,3,4}`. When this object is serialized, the representation in hexadecimal is:

```
ac ed 00 05 73 72 00 11 6a 61 76 61 2e 6c 61 6e
67 2e 49 6e 74 65 67 65 72 12 e2 a0 a4 f7 81 87
38 02 00 01 49 00 05 76 61 6c 75 65 78 72 00 10
6a 61 76 61 2e 6c 61 6e 67 2e 4e 75 6d 62 65 72
86 ac 95 1d 0b 94 e0 8b 02 00 00 78 70 00 00 00
01 73 71 00 7e 00 00 00 00 00 02 73 71 00 7e 00
00 00 00 00 03 73 71 00 7e 00 00 00 00 00 04
```

Figure 3 is a breakdown of what some of the key fields in this serialized object mean. Note that this representation defines the class structure for one serialized integer and then reuses it by reference for the next three integers.

While you will rarely, if ever, work at this level, this will help you understand how serialization actually works. However, an attacker could likely use knowledge of this representation to develop an effective exploit.

After serializing the object, we can later recreate the object in memory by deserializing the encoded representation. Deserializing mirrors the serialization process:

1. Read the first four bytes, checking that the serialization protocol and version number are compatible.
2. Read the following bytes of class metadata (header information), invoke the class loader to create a new instance of the object.
3. Use the class `readObject` method associated with the class to read the integer values to initialize the fields of the object.

Serialization byte representations are typically considered internal implementation details and, as such, either not well documented or not easy to understand. While there are good reasons for hiding the specifics behind the serialization abstraction, this also makes security more difficult. For one thing, there is no clean way to test if an arbitrary byte sequence is or is not a well-defined serialization. Additionally, if any of the serialized bytes are tampered with, the behavior under deserialization is generally undefined, so behavior might differ from platform to platform. These behaviors will inevitably include the potential for harmful consequences that might be exploitable.

`ac ed 00 05` Java serialized object magic number and protocol version number (05).
`73 72 00 11 6a 61 76 61 2e 6c 61 6e 67 2e 49 6e 74 65 67 65 72` Start of new object (73); class description (72); length of the class name (0011); and string for the class name, "`java.lang.Integer`".
`12 e2 a0 a4 f7 81 87 38 02 00 01` Unique identifier for this serialized object (8 bytes); flag to indicate that it supports serialization (02); number of fields in this class (0001).
`49 00 05 76  61 6c 75 65` Field type (49 is "I" for `int`); field name length (0005) and field name "value".
`78` End of data for this class.
`72 00 10 6a 61 76 61 2e 6c 61 6e  67 2e 4e 75 6d 62 65 72` Superclass description (72); length of class name (0010); and string for the class name, "`java.lang.Number`".
`86 ac 95 1d 0b 94 e0 8b 02 00 00` Unique identifier for serialized object (8 bytes); flag to indicate that it supports serialization flag (02); number of fields in this class (0000).
`78 70` End of data for this class (78) and end class hierarchy (70).
`00 00 00 01` The value of the integer associated with this object description (00000001).
`73 71 00 7e 00 00` This is a reference to an object whose definition has appeared previously in this serialization stream (73 71). The object has a handle of 00 7e 00 00. Since the standard defines that handles start at 0x007e0000, this handle is referring to the first object type definition seen (which is everything from the 73 to the 70 above), which describes an integer.
`00 00 00 02` The value of the second integer (00000002).
`73 71 00 7e 00 00` Again, a reference to the first object in the stream, i.e., integer object.
`00 00 00 03` The value of the third integer (00000003).
`73 71 00 7e 00 00` Again, a reference to the first object in the stream, i.e., integer object.
`00 00 00 04` The value of the last integer (00000004).

Figure 3: Explanation of Encoding of Java Serialization Record

14.4 Tampering with Serialized Data

Now that we understand how serialization works at a low level, let's look at what an attacker can do with serialized data.

While the serialized form at first looks like gibberish, if an attacker is able to see the serialized form, they can learn the value the object held at the time of serialization by analyzing the byte stream. So, you need to protect the serialized form of the data as you would protect any form using access controls or encryption.

Suppose an attacker is able to tamper with the serialized byte stream, changing the third value from 3 to 5. The modified value is underlined in red.

```
ac ed 00 05 73 72 00 11 6a 61 76 61 2e 6c 61 6e
67 2e 49 6e 74 65 67 65 72 12 e2 a0 a4 f7 81 87
38 02 00 01 49 00 05 76 61 6c 75 65 78 72 00 10
6a 61 76 61 2e 6c 61 6e 67 2e 4e 75 6d 62 65 72
86 ac 95 1d 0b 94 e0 8b 02 00 00 78 70 00 00 00
01 73 71 00 7e 00 00 00 00 00 02 73 71 00 7e 00
00 00 00 00 05 73 71 00 7e 00 00 00 00 00 04
```

When this byte stream is deserialized, now the third integer will be a value of 5 instead of the original 3.

14.5 A Serialization Exploit in Python

In general, we have to careful when we receive a serialized object from an untrusted source. This means that serialization should never be used as the representation for data from an unknown or untrusted source. We reinforce this point with an example of how just deserializing a serialized object from an untrusted source can cause disastrous results.

The serialization in Python (called *pickling*) works in an interesting, flexible, and somewhat complicated way. The pickled object is actually a tuple, a pair of fields. The first field is a callable object (basically a method name) and the second field is a tuple of the parameters to be passed to that method. This object is sent to the recipient, who deserializes (*unpickles*) it by calling the method, passing it the parameters in the tuple. The result of this call is the deserialized object. This structure allows for a great deal of flexibility when using serialization. In some cases, there can be too much flexibility.

Let us examine an example where a malicious user can create a serialized object that would be damaging to deserialize. At this point, you might stop reading for a moment and think about how you might construct such an example.

Step 1. The client pickles malicious data:

In Python, you can define your own custom serialization (pickling) method. You do this by defining a `__reduce__` method in your class definition. This definition overrides the default pickling method that Python would use.

We are going to create our attack by defining a class with a pickling method that will execute malicious code when it is unpickled. We will do this by creating a (callable object, (parameters)) tuple that will call a dangerous system function with an appropriately dangerous parameter. When the receiver of the pickled object unpickles it, the system function, with the specified parameters, will automatically be called.

In this case, the dangerous system function is `os.system`, which invokes a command shell with a string that specifies the command line to be executed. The command to be executed is something destructive or dangerous like removing all your files.

The new class, called `payload`, is defined on line 1 of Figure *4*, with the custom pickling method defined on lines 2 and 3. Line 5 is code to send a newly created pickled object of class `payload` over a network socket (`soc`).

```
01 class payload(object):
02       def __reduce__(self):
03      return (os.system, ('rm -rf /*',),)
04
05 soc.send(pickle.dumps(payload()))
```

Figure 4: Creating a Class with Malicious Serialization Method and Sending it over a Socket

Step 2. The server unpickles random data:

On line 1 of Figure 5, the receiving process receives the serialized data sent by the client from the socket (`skt`), with 1024 denoting the buffer size. The code then deserializes the receive data on line 2 by calling pickle.`loads` `method` on the serialization data received. The pickle.`loads` method process the data and the malicious tuple from the sender is reconstructed: `(os.system, (‘rm -r /*’,),)`.

```
01 line = skt.recv(1024)
02 obj = pickle.loads(line)
```

Figure 5: Receiving an Object from a Socket and Unpickling It

(Warning: If you try out this code on your own computer, you are likely to do very bad things to it. If you want to test this code, create a new copy of a virtual machine and execute it there. You can also use a safer command like “`ls -l`*”)*

Step 3: Server executes the attack:

Still within `loads` the callable `os.system` is resolved and its arguments tuple is passed to it, resulting in the following system command line being executed: `rm -rf /*`.

Note that for this attack to actually destroy the file system, the server code would need to be executing as root or a similarly privileged account. Running a web server as root is a bad practice in itself – because it listens to requests from the public internet, and thus potentially exposes the entire system to attack. However, even if the server runs at lower privilege, a different tailored attack (e.g., deleting all data to which the server has write access) could be nearly as devastating.

14.6 Using JSON Securely

JSON (JavaScript Object Notation)[4] is widely used as a universal data format on the web that works much like serialization, with the additional advantages of being supported across a wide range of languages and being human-readable as text. JSON is a subset of JavaScript syntax that expresses data as name-value pairs and arrays of values. JSON is designed to be easy to parse and check the validity of the data.

Unlike other forms of serialization described above, JSON is structured data but does not include class metadata itself. Deserializing into an object of a given class requires code that converts the JSON into a constructed object instance. When handling JSON from an untrusted source, this code must also check the validity of the data to ensure it conforms to expected requirements and that all values are of the right type. Always reject data that fails to meet expectations unless reasonable corrections or defaults can be safely provided.

In JavaScript, always use `JSON.stringify`[5] and `JSON.parse`[6] functions to serialize and deserialize text as JSON. While it may be tempting to use the Java native `eval` on JSON text, this should never be done since it potentially executes arbitrary code possibly embedded in the untrusted input in the form of expressions such as function invocations within the JSON. The JSON specification does not allow method calls, but using `eval` would permit them.

---

[4] `http://www.json.org/`

[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`

[6] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse`

14.7 Serialization Risk Mitigations

Serialization is a useful tool, but as we have seen, it must be used with care as the mechanisms behind the implementation can be fragile and hence easily become a source of security problems. Should an attacker manage to tamper with serialized data, the chances of additional problems arising when deserializing spurious data are high.

Serialized data appears opaque and complicated but it can be easily reverse engineered exposing all the contained information to an eavesdropper, as we showed in Section 14.3.

Unless there is certainty that data integrity can be assured, avoiding serialization is the only surefire way of avoiding these issues. With a solid understanding of the risks, if you do want to use serialization, consider applying as many of the following mitigations as applicable.

1. When possible, write a class-specific serialization method that explicitly does not expose sensitive fields or any internal state to the serialization stream. In some cases, it may not be possible to omit sensitive data and still have the object work properly.
2. Ensure that deserialization (including super classes) and object instantiation does not have side effects.
3. **Never deserialize untrusted data.** In general, the behavior of deserialization given arbitrarily tampered data is difficult, if not impossible, to guarantee safeness.
4. Serialized data should be stored securely, protected by access control or signed and encrypted. One useful pattern is for the server to provide signed and encrypted serialization data to a client; later the client can return this information intact to the server where it is processed only after checking the signature.

Always keep in mind that serialized data is just like any other data: it can leak information if exposed, and needs to be handled with care if it originates from an untrusted source.

14.8 Exercises

1. Write code using a standard serialization library to convert a simple data object to a byte sequence and then deserialize to create a clone copy. Test that it works as expected.
2. Extend the code in Exercise 1 to also print out the resulting bytes and see if you can reverse engineer what at least some of the bytes mean. Try serializing different data values to see how the byte sequence changes.

3. Write a new serialization library from scratch (do not use an existing serialization library) that works for integer arrays with two methods. `Serialize(array)` returns a byte sequence representing the array value. `Deserialize(array)` takes a byte sequence from the serialize method and returns an array with the same original values.
4. Extend the code in Exercise 3 to modify the resulting byte sequence before deserializing and see what happens.
    a. Craft a modification that changes a specific element of the array to a new value.
    b. Try modifying the length of the array to be shorter or longer (or even negative).
    c. Looking at the deserialization code, try other modifications to see if you can crash it.
5. Implement mitigations in the code from Exercise 3 and 4 to handle modifications safely. What are the limits of how well you can protect against malicious modification?
6. (Advanced) Write code that serializes a simple data object in a new process so you can handle exceptions including crashes, and fuzz test by randomly perturbing the byte sequence and attempting to deserialize it. What variety of resulting problems can you discover? How fragile is deserialization over a large number of randomized tests?