Introduction to Software Security
Chapter 3.5: Serialization

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Revision 2.2, October 2024.

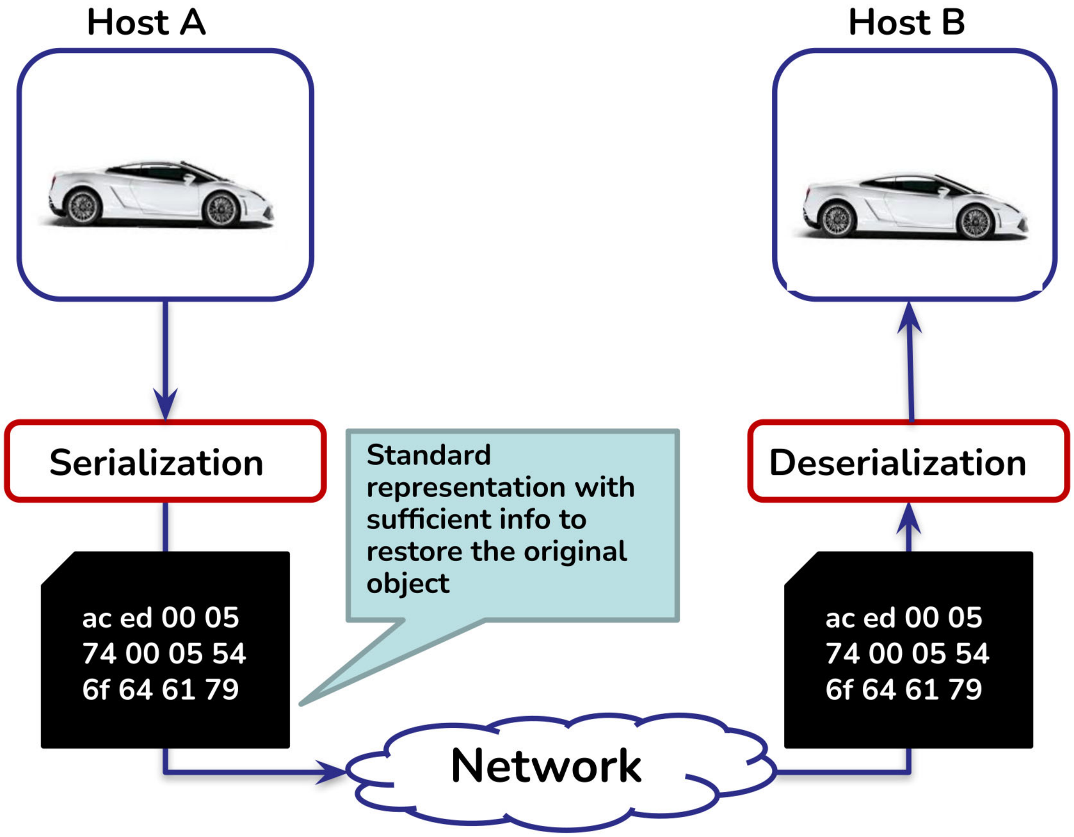
Objectives

- 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.

This module includes examples from both Java and Python, and also mentions other popular language serialization implementations. While the implementation details of serialization differ significantly by language, the underlying principles and fundamental security threats are conceptually similar.

Serialization basics

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 some parts of the state of the program. Serialization is a technique that allows you to package your data objects in a robust and consistent form for storage or transmission, and then later restored to their in-memory form, either on the original machine or a different one. While simple data objects may reliably be represented by the same set of bytes running on similar hardware architectures with compatible software, in general the actual byte representation of objects is not guaranteed for various reasons. As a result, it is inadvisable to store and later reload the same byte contents and expect to get the same object state in the general case. Serialization provides a stable byte representation of the value of software objects that can be sent over a network that potentially will continue to work correctly even in future implementations using different hardware and/or 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 to provide 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.

Perhaps 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 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, assuming the standard library will be fine when it actually does not do the right thing or possibly exposes protected information inadvertently. When custom code handles serialization it needs to avoid all of the usual security pitfalls while expressing and reconstructing properly initialized objects in order for serialization to work properly. 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 often there is 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 or database, 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 (i.e., your own program).

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 that does the serialization) and the receiver (the program that does 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 broken or maliciously modified data will generally result in indeterminate behavior, and that is a ripe opportunity for attackers to craft attacks.

Serialized byte streams are usually incompatible across languages, however at a high level, their structure and form is similar. Serialized data usually begins with a specific header denoting what it is — so as to easily reject random data mistakenly used instead — and often contains a version number allowing future implementation changes while maintaining backward compatibility. Before the object contents are expressed, metadata specifies the class of the data, that at deserialization time allows the runtime to instantiate the correct type of object. Actual field 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 25 bytes is generated that contains sufficient information to express the object metadata and value. It is easy to send these bytes to its peer which then uses the complementary deserializing library to decode the data, determine the correct object type to instantiate, and then initialize it to have an identical value to the original. (A more complete example in full detail appears in the following section.)

This simple example shows the benefits of serialization allowing object state information to cross 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, and that is where security issues arise that need to be mitigated by the implementor (unless a perfectly secure environment can somehow otherwise be assured to rule out any such possibility).

Serialization in various languages

Before considering the security aspects of serialization, we provide a brief overview of how serialization works in several popular languages.

Language	Serializing	Deserializing
Java	Method: writeObject Implemented in:	Method: readObject Implemented in:

	ObjectOutputStream	ObjectInputStream
Python	pickle.dumps(…)	pickle.loads(…)
Ruby	Marshal.dump(…)	Marshal.load(…)

C++ using Boost	boost::archive::text_oarchive oa (filename);
	oa << data;
	Invokes the serialize method. To serialize objects in a user-defined class, you must define a serialize method in that class.
		boost::archive::text_iarchive ia(filename);
		ia >> newdata;
		Invokes the serialize method. User-defined classes handled in the same way as the serialization case.

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

Python uses the standard pickle library to handle serialization: dumps(…) to serialize and loads(…) to deserialize. We will look at an in depth example that shows one form of attack later.

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

C++ Boost serialization uses text archive objects. Serialization writes into an output archive object operating as an output data stream. The **>>** output operator when invoked for class data types calls the class serialize function. Each serialize function uses the **&** operator, or via **>>** recursively serializes nested objects to save or load its data members.

Microsoft Foundation Class (MFC) Library in C++ Visual Studio: Serialization is implemented by classes derived from CObject and overriding the Serialize method. Serialize has a CArchive argument that is used to read and write the object data. The CArchive object has a member function, IsStoring, which indicates whether Serialize is storing (writing data) or loading (reading data).

How serialization works

Let's take a closer look at how Java serialization works on an object containing four integers with values {1, 2, 3, 4}. The serialized byte hexadecimal representation 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
```

Here is a breakdown of what some of the key fields in this serialized object mean:

```
ac ed 00 05
```

Serialization stream magic data header with version number (5).

```
73 72 00 11 6a 61 76 61 2e 6c 61 6e 67 2e 49 6e 74 65 67 65 72
```

Object (73); class description (72); classname length (0011) and string "java.lang.Integer".

```
12 e2 a0 a4 f7 81 87 38 02 00 01
```

Class serial version identifier (8 bytes); supports serialization (02); number of fields (0001).

```
49 00 05 76  61 6c 75 65 78
```

Field type (49 is "I" for Int); field name length (0005) and string “value”; end block data.

```
72 00 10 6a 61 76 61 2e 6c 61 6e  67 2e 4e 75 6d 62 65 72
```

Superclass description (72); classname length (0010) and string "java.lang.Number".

```
86 ac 95 1d 0b 94 e0 8b 02 00 00
```

Class serial version identifier (8 bytes); supports serialization (02); number of fields (0000).

```
78 70
```

End block data (78); end class hierarchy (70).

```
00 00 00 01 73 71 00 7e 00 00
```

The first array value (00000001); object reference (73 71) to handle (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
```

The succeeding array values (2, 3, 4) follow in a similar manner.

After serializing the object, we can later create a clone of the object state by deserializing from that same byte stream. Deserializing mirrors the serialization process:

- Read the first four bytes, checking that the serialization protocol & version number are compatible.
- Read the following bytes of class metadata, invoke the class loader to create a new instance.
- Using the class `readObject` method, read the integer values to initialize the fields of the object.

Serialization byte representations are typically considered internal implementation details and as such not well documented. 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, which inevitably includes the potential for harmful consequences that might be exploitable.

Serialized data tampering

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. Protect the serialized form of sensitive 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.

```
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
```

(The four integer values that comprise the object are in bold, with the modified value underlined.)

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

Python example

Next is an example of how more tampering with serialization provides an attacker more than simply modifying the value of data. The scenario is where a server receives serialized data that it mistakenly trusts, either sent from a malicious client or possibly modified in transit. Perhaps the context is where the server receives some seemingly-innocuous data from the client, and the programmer figured that even if the data itself was incorrect, it was still harmless. As we shall see, by deserializing harmful data, the consequences can be quite bad.

The serialization (*pickling*) in Python works in an interesting, flexible, and somewhat complicated way. The pickled object is actually a pair of fields. The first field is a callable object (basically a method name) and the second 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.

Let’s examine an example where a malicious user can create a serialized object that would be damaging to deserialize. (This is a good point to stop reading and think about how you might construct such an example.)

Step 1. The Client pickles malicious data. Overriding the __reduce__ method, this attack code returns a tuple of a callable method that invokes the system call, and a parameter that contains a string for the UNIX shell command to delete the entire filesystem.

Alternatively, the attacker could perform this serialization privately to capture the serialized byte values needed for the attack, then simply send the resulting byte stream or modify the data on the wire.

(Warning: Obviously, do not try to run this code yourself without appropriate precautions!)

```
class payload(object):
    def __reduce__(self):
    return (os.system, (‘rm –rf /*’,),)

soc.send(pickle.dumps(payload()))
```

Step 2. Server unpickles random data. The server first reads the serialized data sent by the client from the socket (skt) with 1024 denoting the buffer size. Next the code deserializes by invoking loads on the serialization data read. Within loads() the data is parsed, the malicious tuple from the client is reconstructed — (os.system, (‘rm –r /*’,),).

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

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.

Using JSON securely

JSON (JavaScript Object Notation) 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. This subset is designed to be easier 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. 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 completely that fails to meet expectations unless reasonable corrections or defaults can be safely provided.

In JavaScript, always use JSON.stringify() and JSON.parse() functions to serialize and deserialize text as JSON, respectively. While it may be tempting to use 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 function invocations, but nevertheless `eval()` executes them.

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 at a glance appears opaque but it can be easily reverse engineered exposing all the contained information to an eavesdropper, as we showed in the "How serialization works" example above. In the case of complex objects, there is sensitive internal state that appears in the serialized form that is otherwise private. Serialization formats often include metadata or other additional information besides the actual values within an object that may be sensitive.

Unless there is certainty that data integrity can be assured, avoiding serialization is the only surefire way of eluding these potential 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 superclasses) 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 a signed and encrypted serialization blob to a client; later the client can return this intact to the server where it is only processed after signature checking.
5. Sometimes it helps to sanitize deserialized data in a temporary object. For example, deserialize an object first, instantiating and populating it with values, but before actually using the object, ensure

that all fields are reasonable and consistent, or force an error response and destroy any object that appears faulty.

Always keep in mind that serialized data is just like any other data: it can leak information if exposed, and from an untrusted source where it is subject to tampering, it needs to be handled with care. Unless you understand what each byte in the serialized data means and exactly how deserialization will treat the bytes, you should never assume that it will all "just work" in the face of analysis and tampering by a clever attacker.

Serialization is a powerful tool with significant benefits but it also carries inherent security risks. It can be convenient but the machinery of serialization and deserialization adds overhead and complexity that can incur security vulnerability if used improperly. Given the reality that attacks do occur, any use of serialized data incurs some additional risk if tampering ever happens. While there is no magic bullet, the use of multiple mitigations as outlined above can greatly mitigate the risks. The more mitigation and defensive coding you do, the more secure you are.

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 deserializer 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 serialize 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?