JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 8, November-December 2006

Encrypting the Java Serialized Object

Alan Ward, Data-transmission Security Research Group, University of Andorra

Abstract

The cryptographic classes incorporated into Sun's releases of the Java Virtual Machine are often used in e-commerce and other applications. A mechanism exists for signing a Java Class. However, little notice has been given to the possibility of encrypting the Java object instance itself, either for transmission or to maintain state during storage. In this article, we discuss the applicable techniques and benefits of using DES, Triple-DES, Blowfish and AES symmetric secret-key algorithms to encrypt a Java serializeable object using technology available through public means.

1 INTRODUCTION

As the amount of various APIs available for Java grows, so does their use in both Web-based services (applets and servlets) and independent networked and mobile applications: Swing-based client-side programs, EJBs, and more. One of the main factors we think has accelerated this spreading use of Java is the availability for some time now of cryptographic libraries that help ensure security across distributed software platforms.

However, it is one thing to ensure security during transmission, and another altogether to be able to assert that data is not compromised during storage. As has been pointed out in [SMA06], it makes more sense for a hacker to spend time compromising the security of a server that holds several thousands of files, than to eavesdrop a communication line for a potential nugget.

So it is important to pay attention to security measures not just at transmission level, but also to individual files … and perhaps go further, to the level of individual objects.

In this article, the pros and cons of encrypting Java objects are examined, not in terms of Class files that can be signed, transmitted through secure channels (such as SSL) and stored in protected volumes on disk, but rather as serialized objects that can do all that – and maintain their state through transmission as well. The fact they have a smaller size in bytes was not an issue.

The test series used was centered on statistical measures of both the original serialized object and the encrypted streams with symmetric cryptography. After reviewing the technological choices made, the results thereof and a short comparison with the assymetric RSA algorithm are presented.

Cite this article as follows: Alan Ward: "Encrypting the Java Serialized Object", in *Journal of Object Technology*, vol. 5, no. 8, November - December 2006, pp. 49-57 http://www.jot.fm/issues/issue_2006_11/column6

2 OBJECTIVES AND DESIGN

The main purpose of this series of tests was to study the particularities of the serialized Java object as regards being encrypted. Specifically, it was decided to use stream cryptography, since using the combination of Java's Object and Cipher Streams seemed interesting.

Specific questions posed were:

- Can a serialized Java object stream be encrypted & decrypted without loss of state?
- What are the specificities of Java serialized objects are regards cryptography?
- Which symmetric algorithms give more speed – and are the differences between algorithms significant?
- Which symmetric algorithms – among the more standard – hide best the nature of the object transmitted?

As for the choice of symmetrical cryptographic algorithms versus public/private key schemes, it was noted very early on that asymmetrical cryptography is expensive in terms of computing time. So when public/private key algorithms are used, they are currently used only in an initial stage of the communication process, in order to exchange a once-off symmetric cryptographic key. This is then used for the bulk of transmission.

For example, during an SSL session, an RSA public certificate is used to connect to the server. A symmetric key is then generated, and the rest of the communication for this session only is encrypted with this second key, often using the DES algorithm.

For this reason, it was decided to concentrate on some of the most widespread symmetrical algorithms, including DES, 3-DES (Triple-DES), Blowfish and AES (all included in the Sun cryptographic Provider). A final test was done with the RSA algorithm, but further research should be done on the asymmetric protocol side.

3 TECHNOLOGICAL ASPECTS

Two technological aspects were studied in more detail are regards the implementation of encrypting a serialized object. One was the structure of the Java object itself, in serialized form: i.e. the byte sequence produced when an object is serialized and sent through an ObjectOutputStream. This differs significantly from the structure found in the Class file structure on disk.

The second aspect was the treatment of bytes by the Java system of Streams which, as is noted in Sun's JDK documentation [Sun04], still contains several interesting features.

Finally, a choice had to be made between the several dozen symmetric and asymmetric cryptographic algorithms available, either through Sun's own cryptographic provider or other plug-in providers such as BouncyCastle (www.bouncycastle.org).

The Java Serialized Object

A modified filtering Stream class was used to collate data on the distribution of ASCII character codes, byte values, on-line in the Stream as several different objects were serialized and sent through. Results are graphed in Figure 1. Codes are grouped by 16-value intervals for ease of visualization.

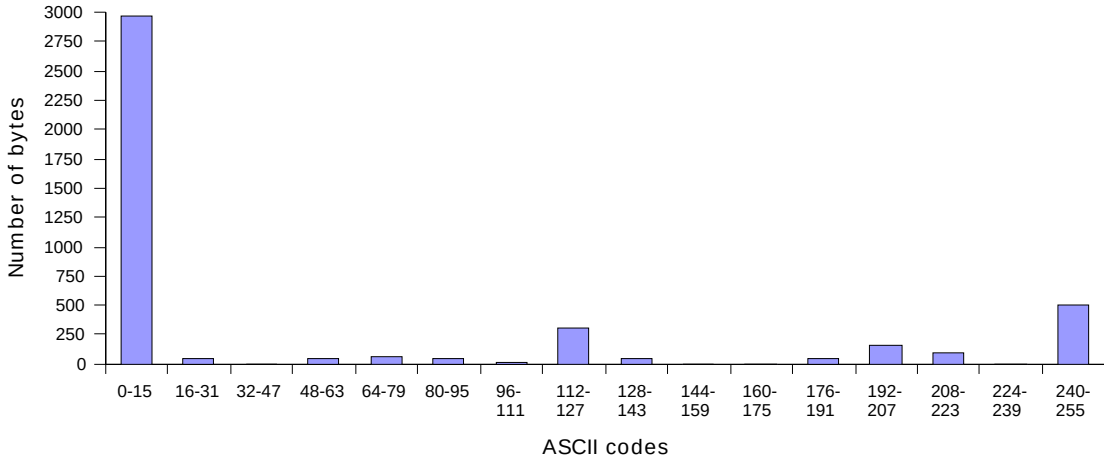


Figure 1. Serialized object character code distribution

As can be seen, a very large proportion of code values are found in the first group. In fact, "0" values occur as more than 60% of all bytes transmitted, implying a high proportion of null two- and four-bytes sequences as well.

Further tests show that this proportion varies slightly across the classes transported, but holds in general. The only exception found so far are objects containing large amounts of character or string data hard coded into variable initialization sentences.

This is naturally not good in cryptographic terms, as it could leave encrypted object streams open to frequency analysis attacks.

Working with Streams

Sun's JDK documentation [Sun04] indicates String data is now all encoded in a "slight modification of the Unicode Standard". While this is not a bad idea *per se*, it complicates our task as the String is a convenient way of storing and moving around serialized objects, and one we wanted to use. However, under UTF-8:

- Characters with byte values between 0 and 127 are transmitted as-is.
- In all other two- or three-byte character codes, the first byte has its most important bit set to 1. Thus a byte value of 128 to 255 can only occur as a first byte (or subsequent byte) of a multi-byte character code.

A short test sequence showed us that in pragmatic terms, we are not about to get away with sending any byte value above 128 through the original assembly of Streams, shown in Figure 2:

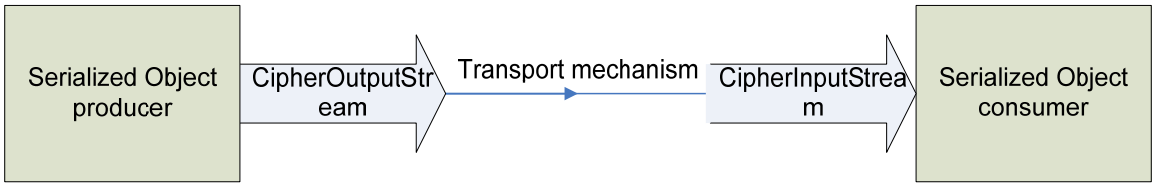


Figure 2. Initial transmission scheme.

A means was then devised to by-pass this problem by inserting a pair of converting Streams, the first one (ConverterOutputStream) converting the serialized object's bytes to MIME base-64 encoding, and the other (ConvertedInputStream) reconverting base-64-encoded bytes to the original byte values in the 0 to 255 range.

These converter Streams were then modified to be used as a data-gathering instrument as well, drawing up transmission statistics at a single point during the transmission process.

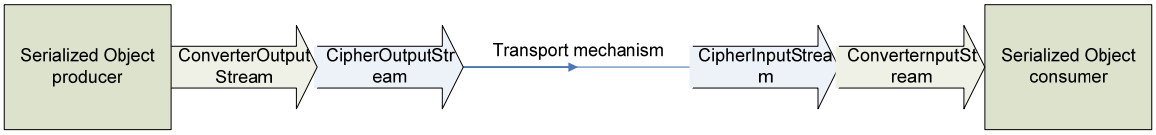


Figure 3. Modified transmission architecture.

In this figure, the producer can be the original producer of the object, a transport emission point, a data retrieval system or any other serialized object source. The consumer can likewise be the final user of the object, a transport reception point, data storage system or another object sink point. During transport, encrypted objects may be placed in short- or long-term storage, or replicated – while maintaining their encryptation throughout.

The code used to convert a serialized object into a String is fairly neat:

```
MyObject myObject = new MyObject();
   // MyObject implements Serializable
String result;

ByteArrayOutputStream bout = new ByteArrayOutputStream();
CipherOutputStream cout =
   new CipherOutputStream(bout, cipher);
ConvertOutputStream convout = new ConvertOutputStream(cout);
ObjectOutputStream oout = new ObjectOutputStream(convout);

oout.writeObject(myObject); // repeat as needed
oout.close();
result = bout.toString();
```

Reception of the object at the other end is an analogous process.

4 TEST RESULT EVALUATION

The following data has been obtained by sending a total of 100 different objects, amounting to a total about 4 Kbytes of data, through the system. Time includes object creation, both encryption and decryption, as well as a final checkup on each object after its' being decrypted. The platform used was a Mobile Pentium IV at 2.4 GHz, running Sun's JDK version 1.5 under Linux.

Speed of the encryptation / decryptation process

Number of objects	DES stream	3-DES stream	Blowfish	AES
Key length (bits)	*52*	*168*	*128*	*128*
100	2321	2319	2516	2341
200	3175	3056	3215	3080
400	4186	4193	4345	4405

Table 1. Algorithm times (in ms) for different numbers of objects

It should be noted that neither keys nor the objects themselves were stored on disk during this process. The very small differences in duration for each test, as well as subsequent profiling, lead to believe that most time is spent in key generation, object creation, testing, and general housekeeping. The time spent in actual encrypting and decrypting is a small fraction of the total.

This result is confirmed by statistical linear regression, which shows the very good correlation expected:

	DES stream	3-DES stream	Blowfish	AES
Slope	6.05	6.17	6.03	6.84
Intercept	1815.5	1750.5	1951	1678.5
Correlation	0.990	0.998	0.998	1.000

Table 2. Slope = time (in ms) per object (creation, encrypting, decrypting and testing).
Intercept = time (in ms) for system setup, key generation and "housekeeping"

Attention is drawn to two specific points. In the first place, the difference in time per object for each type of algorithm is not significant, even though their inner workings are rather dissimilar (consider DES and Triple-DES).

On the other hand, the differences in setup time can be attributed mainly to the key generation process, all other steps being identical – but the time spent is not at all proportional to actual key length in bits.

Statistical analysis of the encrypted streams

Since the results of the statistical analysis of serialized objects themselves was not good, it remained to be seen if the algorithms were capable of introducing more entropy into the encrypted streams.

In an ideal encrypted stream, each possible byte code would appear with the exact average value of 1/256 – following the normal distribution -, and the standard deviation of the whole would be null. Results for the encrypted object streams were:

	Serialized object	DES stream	3-DES stream	Blowfish	AES
Average	0.0039	0.0039	0.0039	0.0039	0.0039
Std. Deviation	0.0385	0.0075	0.0069	0.0068	0.0028

Table 3. Statistics for each algorithm used

In the first place, we are reassured that all cryptographic algorithms distribute byte-values across the range, giving much "better" (i.e. lower) standard deviations than the original stream. The these results also confirm what is common knowledge: i.e. that Triple-DES is better than DES, and the other two algorithms are better than either of them in terms of security.

More surprising was that, with equal-length 128-bit keys, AES outperformed Blowfish to such an extent.

This can be seen more easily in Figure 4, showing the occurences of each byte code grouped by 16-value intervals as before. The results of the AES-encoded stream approximates rather well the ideal flat line.

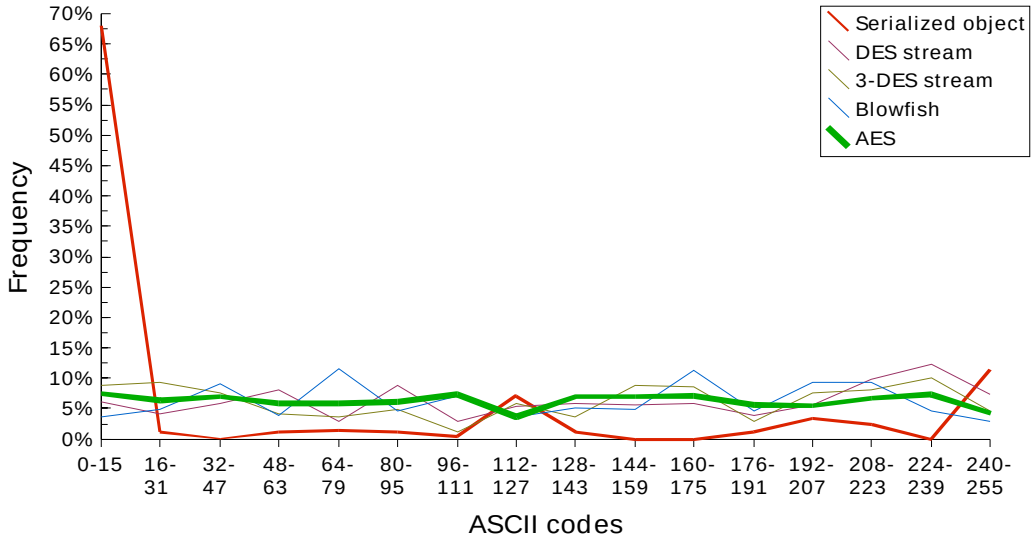


Figure 4. Character code distribution comparison:
original class file and various encrypted data streams. Flattest distribution is best!

Comparing the symmetric algorithms to asymmetric RSA

Finally, a similar test was made with RSA public/private key cryptography. In this case, our main problem was combining Cipher and Object streams: no bytes were transmitted when the RSA cipher was used. Further analysis showed that the problem arose from the process of separating data bytes into blocks:

- RSA block-cipher requires data to be separated into blocks of up to 117 bytes at most, when 1024-bit keys are used. Encyphered output is twice the size of input.
- The RSA Cipher object was unable to do this correctly.
- DES and the other algorithms performing CBC or ECB encryptation need a similar operation, but Cipher performs correctly in their case.

To get around this bug (feature?), we devised a wrapper around the Cipher update() and doFinal() methods that broke down the serialized Java object stream into 64-byte chunks, and padded the end up to a 64-byte boundary. We thus replicated the behaviour expected of - but not obtained from - the Cipher object instanciated with:

```
Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
```

As regards time efficiency, results are much as espected: RSA is slower, albeight not very much so if we consider that key lengths are also much longer. It is interesting to note, however, that efficiency grows with key size. We used 32-byte blocks with 512-bit keys and 64-byte blocks with 1024-bit keys, giving a slope of 51.96 ms/object for the shorter key and only 45.03 ms/object for the longer:

Number of objects	AES	RSA	RSA
Key length (bits)	*128*	*512*	*1024*
100	2341	3021	3925
200	3080	6030	7096
400	4405	18171	17168

Table 4: Times (in ms) for the symmetric-key AES algorithm and assymmetric RSA.

Times are longer, but the number of bytes emitted is also twice as large as with symmetric algorithms: this is not good as regards security, as a potential attack has more data to work with.

On the other hand, standard deviation of the resulting byte-stream values is down to 0.0009 – very much better than the 0.0028 AES best value for symmetric algorithms. Distribution curves reflect this fact, giving a flatter curve for RSA:

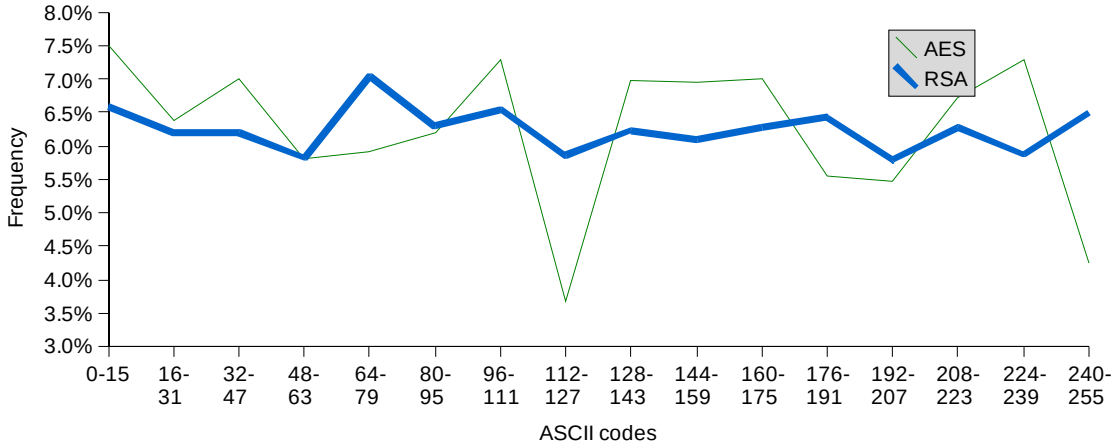


Figure 5. Comparison of AES and RSA byte freqencies in crypted object streams.

5 CONCLUSIONS AND FURTHER REMARKS

With this series of tests it has been shown that Java serialized objects can go through a process a encryptation and decryptation in the form of Cipher Streams, and survive.

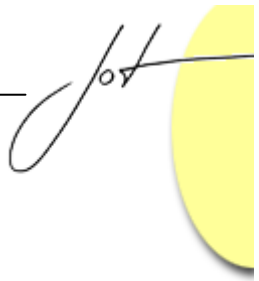
It has also been shown that symmetric key cryptography algorithms are rather better implemented than assymetric RSA in Sun's JDK, as regards ease of use with Streams.

On the other hand, and concerning only the specific test object set used, the AES algorithm outperformed the other three symmetric cryptographic series as regards diguising the Java object stream's unbalanced byte value distribution. It gave good performace figures as well.

Finally, it can be confirmed that assymetric RSA encryptation is about twice as slow as symmetric encryptation, and produces larger encrypted data streams, but on the other hand it also produces much flatter byte distributions. This may be a conclusive factor when cryptography needs to be used.

However, these results have been obtained with a small number of non-functional objects, designed with the aim of representing the average behaviour of most applications. They would need to be confirmed by further statistical studies on real-world classes used in commercial or scientific applications, and possibly containing hard-coded data forming particular distribution figures.

At the same time, other cryptographic algorithms, whether included or not in the basic Provider, are now being used in real-life applications. An example is assymetric public/private key cryptography with the Elliptic Curve algorithm, that opens new and interesting perspectives that should be compared both to symmetric and to RSA schemes.



REFERENCES

[APS99] George Apostopoulos, Vinod Peris, Debanjan Saha: *Transport Layer Security: How much does it really cost*?, INFOCOM '99, IEEE, 1999.

[CMS04] Ajay Chander, Joh C. Mitchell, Insik Shin: *Mobile Code Security by Java Bytecode Instrumentation*, Software Engineering and Applications 2004, MIT Cambridge, USA, 2004.

[RoJS01] Volker Roth, Mehrdad Jalali-Sohi: *Concepts and Architecture of a Security-Centric Mobile Agent Server*, Fifth International Symposium on Autonomous Decentralized Systems, IEEE, 2001.

[Roth02] Volker Roth: *Java Security Architecture and Extension in Practice*, Dr. Dobbs Journal, 2002(335), 2002.

[Shm] Anat Sarig Shmueli: *Java Security*, IBM Haifa Research Lab.

[SMA06] Ed Simon, Paul Madsen, Carlisle Adams: *An Introduction to XML Digital Signatures*, O'Reilly Media Inc., available on-line at http://www.xml.com/, 2006

[StL00] Andreas Sterbenz, Peter Lipp: *Performace of the AES Candidate Algorithms in Java*, AES Candidate Conference, 2000.

[Sun99] Sun Microsystems, Inc. (Santa Clara, CA, USA): *Java™ Cryptography Architecture API Specification and Reference*, available on-line at http://java.sun.com, 1999.

[Sun02] Sun Microsystems, Inc. (Santa Clara, CA, USA): *Java™ Cryptography Extension (JCE) Reference Guide*, available on-line at http://java.sun.com, 1996-2002.

[Sun04] Sun Microsystems, Inc. (Santa Clara, CA, USA): *JDK™ 5.0 Documentation*, available on-line at http://java.sun.com, 2004.

[Vil98] Antti Viljamaa, Jukka Viljamaa: *Java and Internet Security*, Report C-1998-45, University of Helsinki, 1998.

About the author

Alan Ward is an associate professor at the University of Andorra (Principality of Andorra, Europe). He can be reached at award[AT]uda.ad.