TDDC03 Projects, Spring 2006

Java Permissions -- Are they all created equal?

Pierre-Emmanuel Bourgué

Jean-Sébastien Susset

Supervisor: Almut Herzog

Java Permissions -- Are they all created equal?

Pierre-Emmanuel Bourgué Jean-Sébastien Susset
Linköpings universitetet, Sweden
Email: {piebo882, jeasu165}@student.liu.se

Abstract

The paper concerns the classification of Java permissions according to their severity. It provides both technical comments and tips for average users, and also a collection of codes showing the dangerousness of these permissions. Finally, the classification gives good support to compare the various permissions

1. Introduction

In Java, some sensitive actions may require special rights, called permissions, from the user in order to be executed. Applets always need these permissions, but applications only need them if the user has asked for a high level of security (SecurityManager enabled).

The difficulty for a user is to know which permissions are necessary so that a given application can run, what a given permission really allows to the code, what are the risks for the user's system when granting a given permission to a code…etc.

In order to help users, we rank and comment all the Java permissions both with technical details and user-friendly hints explaining what the permission allows the code to do. Finally, we have implemented code examples showing these risks in a more concrete manner by exploiting security flaws created when granting permissions.

2. Background

Here are some reminders about security concepts which are used afterwards.

Bishop [1] defines the three following principles of security:

- *Confidentiality* is the concealment of information or resources.
- *Integrity* refers to the trustworthiness of data or resources, and it is usually phrased in terms of preventing improper or unauthorized change.
- *Availability* refers to the ability to use the information or resource desired.

DoS : Attempt to block availability [1].

According to the Java website [2]:

- A p*ermission* represents access to a system resource.
- *Security policy file*: The policy file(s) specify what permissions are allowed for code from specified code sources.

3. Solution and Analysis

Java permission can have at most two parameters: a target name and optionally a list of one or several actions.
Here is an example for FilePermission:

```
permission java.io.FilePermission "/tmp/abc",
"read";
```

There are two types of permissions:

- Predefined permissions (Java2 Permissions) that we worked on;
- User-defined permissions i.e. customized permissions developed for a particular goal.

One can create a new permission by extending the Permission class or one of its subclasses (BasicPermission, SocketPermission…etc.). A user-defined permission can be added to the policy file as any other native permission. Sometimes, the class of this permission is loaded at runtime from a distant host, so when the security policy is initialized the permission is "unresolved". In this case, the UnresolvedPermission is needed.

The user-defined permissions are at least as dangerous as the permissions they extend. First, there are risks due to the parent permissions (cf. classification). Second, there are new threats due to the re-implementation, possibly by a hacker, of native methods and the development of new methods, possibly malicious.

The enclosed documents constitute the completion of our work. It contains a technical comment and a user-friendly hint for each permission/target name. This tip is intended to help an average user so that he can take the decision to grant or not the given permission. The technical comment especially explains the ways a malicious code could exploit the permissions to improve its attack (directly or indirectly).

The methodology we used in the exploit code development is to show the execution of a single code in two cases: when the policy file is empty and when the given permission (and possibly others if it is really necessary for the demonstration) is granted.

The code simulates a typical attack that hacker could perform exploiting the given permission.

3.1. Evaluation

Classify the Java permissions by their severity is a bit tricky. Indeed, many parameters have to be taken into account:

- What are the priorities in the user's security policy? It can be confidentiality, integrity and/or availability. However, it is often impossible to enforce them simultaneously. So, a choice must be done by the user (person or company) depending on what he thinks to be the worst for him/his company. It may be DoS, the leak of confidential data, or the corruption of data;

For example, if a company gives top priority to the integrity and availability of data at the expense of the confidentiality, the security policy will be very strict on permissions which may cause DoS but will be looser on those which may facilitate leak of data.

- From one people to another, the definition of a level of gravity varies a lot. It is quite a subjective feeling. Where does high level of severity finish and where does medium level of severity begin?

- The consequences of granting a Java permission often depend on the granting or not of other permissions. The number of possible scenario is as huge as the number of permissions, not counting the user-defined permissions.

In each level of severity, the Java permissions cause the three criteria to be no longer maintained. The severity of this flaw is assessed at a level according to the type (common, confidential, secret…etc.) and the quantity/range of data that can be accessed or corrupted (disclosure of information), the damage and consequences of the attack (e.g. DoS) on the system/user's business …etc.

We classified permissions into three different levels of dangerousness: high, medium and low severity.

High:

- Those which may obtain directly or indirectly all the privileges or at least a lot more than they should have had;
- Those which can reach critical data and methods without restrictions, e.g. protected variables, native classes or external libraries;
- Those which can do a important DoS (possibly tricky and silent), by controlling the user's data and resources and damaging the user's system;
- Those which can retrieve very critical and secret data that can possibly be re-used in future attacks;
- Those which are necessary for the other permissions to be really dangerous, e.g. SocketPermission;
- Those which severely fool the user and may cause important repercussions in the future.

Medium:

- Those which cause quite important but obvious DoS (the user will be able to fix the problem or the DoS will probably not be able to damage the system very long);
- Those which can fool the user in a smaller proportion or less slyly than in the highest level;
- Those which can modify properties of the security configuration, but which cannot obtain more permissions anyway;
- Those which can retrieve information about the security policy that can be useful to prepare a future attack;
- Those which can access the user's data without being able to change it;
- Those which can obviously retrieve confidential data.

Low:

- Those which can do small DoS;
- Those which can retrieve basic system properties without being able to use them immediately;
- Those which can hide the hacker's traces (Logging).

4. Conclusions

Our classification contains user-friendly hints whose goal is to explain in a simple and colourful way to users the risks to grant permissions to codes. These tips could be used for this purpose in a user interface.

Besides, the classification gives an overview of the severity of each action according to its threats for the user, thus permitting a comparison between the permissions.

5. References

[1] Matt Bishop, Introduction to computer security, Addison-Wesley, 2004.

[2] Permissions in the Java TM 2 Standard Edition Development Kit (JDK) *http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html*, spring 2006

High severity

java.security.AllPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
none	Grant the code with all the permissions. That will help the code a lot to pirate the user's system. See below to have details about the other permissions. So, Confidentiality, Integrity and Availability are no longer ensured.	Granting this permission allows the code to do what it wants on your system.	The exploit code displays one system property (os.name) and creates a file.

java.lang.RuntimePermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
createClassLoader	Grant this permission gives ALL the permissions to the code. Indeed, a malicious code could make use ClassLoader to give it all the rights. So, Confidentiality, Integrity and Availability are no longer ensured.	Granting this permission may allow a hacker to grant himself all the rights he needs before attacking your system.	An exploit strategy would be to create our own subclass of ClassLoader and then use it to load malicious classes granted with adequate permissions.
createSecurityManager	Granting this permission allows to create a new SecurityManager and use it to replace the former one. The	Granting this permission allows a malicious code to change your security policy. This is very	The exploit code uses a system manager that does not check permissions before the execution of

	new SecurityManager can be malicious and bypass the check permission mechanism. If the security manager does not check the permissions, it has the same consequences than granting "All permission". So, Confidentiality, Integrity and Availability are no longer ensured.	dangerous because a hacker could grant himself privileges in order to do more damage on your computer.	a file.
setSecurityManager	Granting this permission allows setting the SecurityManager, replacing the former one. The SecurityManager can be malicious and bypass the check permission mechanism. If the security manager does not check the permissions, it has the same consequences than granting "All permission". So, Confidentiality, Integrity and Availability are no longer ensured.	Granting this permission allows a malicious code to change your security policy. This is very dangerous because a hacker could grant himself privileges in order to do more damage on your computer.	Not implemented.
loadLibrary.{library name}	Allow the code to load native code libraries. Because they do not prevent malicious behaviour at this level, a malicious code may use this opportunity to pirate the user's system. For instance, a malicious code may use native methods to explore the user's computer, to make the system misbehave or simply corrupt existing data. So, Confidentiality, Integrity and	Granting this permission allows the code to access powerful but unsafe system operations that could damage your computer.	An exploit strategy would be to load a native library (.so on a Unix station and .dll on a Windows station) and use its methods to launch the execution of a program or disturb the system by destroying some processes.

	Availability are no longer ensured. Threats: espionage and DoS.		
accessClassInPackage. {package name}	Allow the code to load classes from packages that it normally can not reach. These classes could help a malicious code to attack the user's system. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: espionage and DoS.	Granting this permission allows the code to access powerful but unsafe operations that could damage your computer.	Not implemented.
defineClassInPackage. {package name}	Allow the code to add new classes in any package. As system packages have special rights, adding a class in such a package will also give the malicious class these permissions and help the code to attack to attack the user's system. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: espionage and DoS.	Granting this permission allows the code to define new operations, possibly malicious, in some places that your system considers as trustworthy. This is dangerous because your system will then trust the hacker's operations and do not check their severity.	Not implemented.
accessDeclaredMembers	Allow the code to access variables and methods regardless of their protection level (public, protected or private. So, a malicious code may access private variable of well-known classes and bypass the protections, e.g. allowed values. This can help a hacker to pirate the user's system in a better way. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: DoS and espionage.	Granting this permission allows the code to access data and operations in the application without any restrictions. This could cause the application to misbehave.	A possible exploit strategy could be to access protected variables and to change their values regardless of the original restrictions (via accessors). Cf. ReflectPermission.

getClassLoader	Allow the code to retrieve the class loader of the calling class. A malicious code can then load forbidden classes (distant classes for instance) that is classes that this ClassLoader also manages. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: DoS and espionage.	Granting this permission allows a hacker to use some system actions that he should never have the right to access. A hacker could use such operations to do more damage on your system.	Not implemented.
setContextClassLoader	Allow the code to change "which ContextClassLoader is used for a particular thread, including system threads". So, Confidentiality, Integrity and Availability are no longer ensured. Threats: DoS and espionage.	Granting this permission allows a hacker to use some system actions that he should never have the right to access. A hacker could use such operations to do more damage on your system.	Not implemented.
setIO	Allow to route the information from the standard input, output and error streams to other destinations. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: Espionage, fool user and DoS.	Granting this permission may allow a hacker to control the input (keyboard) and output (screen or file) of your computer. This is dangerous because a hacker could spy your inputs and deny you access to the screen, for example.	Possible exploit strategies would be for instance: - to deny the user access to the input and output of his computer by routing them in hidden destinations. - to steal confidential information (password, industrial secrets…) from the input.
writeFileDescriptor	Allow the code to access FileDescriptors and to write in the associated files. A malicious code may insert a huge amount of data (possibly corrupted by viruses) in the user's disk. So, Confidentiality, Integrity and Availability are no longer ensured.	Granting this permission allows an attacker to write into your files or other communication channels like your internet connections. This is very dangerous because an attacker may be able to write viruses on your disk.	A possible exploit code would be to write into a file, via its file descriptor, a huge amount of data in order to overload the user's disk.

	Threats: Espionage and DoS.		

java.security.SecurityPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
setPolicy	A security policy states which permissions are granted, to who and in which situations. This permission allows the code to replace the existing security policy by another one. A malicious code may create its own security policy and grant itself all the permissions in order to make the maximum possible damage on the user's system. So, Confidentiality, Integrity and Availability are not granted anymore. Threats: espionage, fool users, DoS.	Granting this permission allows the code to grant itself the privileges it needs. This is very dangerous because malicious code may use that to make damage on your computer.	Not implemented
setSystemScope	Allow to change the system scope. A system scope is the mapping between trustworthy real world object (people, company) and their own public key. It can add certificates that the system should not be trusted. Integrity and confidentiality and availability are no longer ensured.	Granting this permission allows your application to execute untrustworthy code by artificially granting them a certificate. Hackers' attack can be facilitated by granting this permission.	Not implemented

	Threats: Espionage, fool users, DoS.		
setSignerKeyPair	A malicious code can change users' keys pair with weaker pair. So, it can use it to eavesdrop more easily data communication. The target of this attack is the confidentiality of the user's data.	Allow this permission can endanger the confidentiality of your signed messages. Hackers can eavesdrop more easily your transmission.	Not implemented
setIdentityPublicKey	Allow the code to modify the public key of a trusted Identity (person or company). A malicious code may replace the current public key by its own one. So, applets or applications signed by the hacker's private key will obtain all the permissions granted for the former trusted Identity. The hacker may use them to damage the system or spy the user's data. So, Confidentiality, Integrity and availability are no longer ensured. Threats: espionage, DoS and fool users.	Granting this permission allows the code to usurp the identity of someone you trust by another identity, possibly malicious. Thus, you may be fool by a hacker, thinking he is a trustworthy person.	Not implemented
addIdentityCertificate	Allow the code to add a new certificate to a given Identity. If it is a trusty Identity, with a known public key, the certificate must contain the same public key. Otherwise, the public key in the new certificate will from then also be the Identity's public key (cf. setIdentityPublicKey possible	Granting this permission allows the code to add a new certificate, possibly created by a hacker, to an existing contact. This could give the hacker your trust.	Not implemented.

	damage threats). A malicious code may prepare a future attack with a signed applet or application using this certificate and use the corresponding privileges to pirate the system in a better way. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: Espionage, fool users, DoS.		
getSignerPrivateKey	A malicious code can retrieve the private key of the current user. If you allow this permission to a malicious server, the code can send the key and hackers can use it to usurp your identity and decrypt your confidential data. So, the integrity availability and confidentiality is no longer ensured because a private key can be use to decrypt cipher from public key and sign your data. Threat : DoS, espionage	If you grant this permission, malicious code can usurp your identity and decrypt your confidential data.	Not implemented
removeIdentityCertificate	Allow the code to remove a certificate from an Identity (person or company). A malicious code may eliminate a certificate to deny access or diminish the privileges of an Identity. So, Integrity and Availability are no	Granting this permission allows the code to remove an existing certificate proving your trust in a given contact. Thus, misbehaviours may happen in your future transactions.	Not implemented

	longer ensured. Threats: DoS.		

java.io.FilePermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
execute	Allow to execute a file. Granting this permission allows the code to launch other softwares without advising you. These softwares can be malicious. So, Confidentiality, Integrity and Availability are no longer ensured, depending on the nature of the executable file.	Granting this permission allows the code to launch other softwares without any advice. If this software are malicious your system can be damaged.	The exploit code executes a chosen file.
write	Allow to create and modify files. If a malicious code gets this permission, it can empty file or create malicious files. So, Integrity of files is not ensured.	Granting this permission allows the code to modify your files and create new ones in your system. If your system files are modified, your computer can misbehave.	The exploit code creates a file on the computer.
delete	Allow to delete files. So, Availability is no longer ensured. Threat: DoS.	Granting this permission allows the code to delete files on your system.	The exploit code deletes a given file.

java.net.SocketPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
connect	Allow the code to create a connection with a given server (or every server if one use the wildcard *). Then, the code may send over this socket some confidential data. It can also receive malicious data from the distant host. However, in order to make damage, the hacker may need other permissions (FilePermission, PropertyPermission...). So, Confidentiality and Availability are no longer ensured. Threats: DoS and espionage (at least).	Granting this permission allows the code to connect to a distant computer. This is dangerous because the code may connect you to a hacker's server.	The exploit code (client-side) creates a Socket on a distant host (server-side). Then, it sends confidential data (a String) and wait for the answer of the server (a String).
accept	Allow the code to accept the creation of a socket and a connection from a distant host:port. A malicious distant host may then send and receive information from the user's host. So, if the hacker has collected confidential data before (thanks to other permissions like PropertyPermission, FilePermission…etc.), he will be able to send it on his own host. So, Confidentiality and Availability are no longer ensured.	Granting this permission allows the code to accept connection from a distant computer. This is dangerous because the code may accept a connection from a hacker's server.	A server-side program waits for connections on a given port. It accepts the connection from the client-side program. This client takes advantage of the server by sending useless data and wasting its time. It is an example of DoS. This would be even worse if the server was concurrent because many connections would be able to do this kind of DoS in the same time, overloading the server.

	Threats: espionage and DoS (at least).		

java.awt.AWTPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
listenToAllAWTEvents	Allow the code to be notified of all the graphics events on the computer. A malicious code may be able to : - retrieve some confidential information, e.g. input from the keyboard. - Modify current graphics events into process (AWTQueue) - Stop the normal use of the computer, e.g. by adding random graphics events So Confidentiality, Integrity and Availability are not ensured anymore. Threats: espionage and DoS.	Granting this permission allows the code to spy and control what you do with your mouse and keyboard. So, it may steal your passwords.	A exploit strategy would be to add an AWTEventListener to the Toolkit. Then, the code can choose what type of graphics events it wants to listen (key event, cursor's moving, frame's moving…etc). For instance, the code will be able to retrieve each character entered in a password field, thus obtaining the user's password.
accessClipboard	Allow the code to have a read and write access to the AWT Clipboard. A malicious code may see the content of a file you copy/paste for example. It can also corrupt or erase the data.	Granting this permission allows the code to access the data you copy/cut (file, text…). Thus, you can lose data that you cut/copied in the clipboard if a hacker exploits this permission.	The exploit code scans the clipboard and shows the first content being a String or an InputStreamReader. If you copy a portion of text in a document and then run the exploit code, the copied

	So Confidentiality, Integrity and Availability are not ensured anymore. Threats: espionage and DoS.		data will be displayed in the console.
readDisplayPixels	Allow the code to read pixels from the screen. A malicious code may be able to look at the user's screen and possibly be aware of confidential information. Confidentiality is not ensured anymore. Threats: espionage.	Granting this permission allows the code to watch your screen and thus spy what you are doing.	Not implemented.

java.lang.reflect.ReflectPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
suppressAccessChecks	Allow the code to access variable and method regardless of their protection level (public, protected or private). It also permits a code to modify a “final” variable. So, a malicious code may access private variables of well-known classes and bypass the protections, e.g. allowed values. This can help a hacker to pirate the user’s system in a better way. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: DoS (at least)	Granting this permission allows the code to access data without any restrictions.	The exploit code shows that, thanks to this permission, it is possible to access a private variable without passing by its accessors (setXX and getXX), and so, set value that is normally not allowed. In this case, the variable is initialized at 50. The code set it to -3 when the values are normally restricted by the accessor setXX between 0 and 100.

java.net.NetPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
specifyStreamHandler	Allow the code to create an URL instance and specify its handler. The handler handles the connection mechanism for a given protocol. A malicious code may create its own handler for a given protocol and use it to have an easiest access to some data that it should never have access otherwise. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: espionage and DoS.	Granting this permission allows the code to redefine the way you communicate and access data (http, ftp, file…). So, a hacker may be able to access confidential data or make your communication misbehave.	Not implemented

javax.net.ssl.SSLPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
setHostnameVerifier	A supposed trusty and secure connection may become untrustworthy and insecure if such permission is given to a malicious code. Indeed, if there is a mismatch between a server (contacted via an SSL connection) and its name in the certificate, the code may simply authorize the connection. So, if the hacker owns this server, he may be able to use the privileges granted by the certificate to damage the user's system. So, Integrity is no longer ensured (also possibly Confidentiality and Availability, depending on the certificate). Threats: fool users (at least).	Granting this permission allows the code to bypass a so-called secure connection and so to connect your computer to a possibly malicious distant computer. Basically, this permission authorizes a hacker to fool the security protections.	The exploit code creates a HostVerifier whose verification method always return 'true'. So, during the secure connection handshake, a mismatch between the name in the certificate and the host will always be authorized.

Medium severity

java.lang.RuntimePermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
setFactory	Allow the code to create its own implementation of factories (SocketFactory, StreamHandlerFactory …) These factories are used to create instances of the corresponding classes. So a malicious code could implement factories which creates modified instances (in comparison with the original classes - SocketFactory…) by adding additional operations or by corrupting the existing ones. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: DoS and espionage.	Granting this permission may allow a hacker to redefine communication mechanisms used for network connections for instance. A hacker may use this opportunity to snoop on the data transferred or corrupt it.	An exploit strategy would be to create a new SocketFactory that can create a new subclass of Socket. This class may deal more loosely with security, network parameters, e.g. timeout, or compression for instance. The malicious code could use this to do DoS on the user's connections (random modification of the timeout for example) or to copy the data transferred and send it on a distant host via another socket.
readFileDescriptor	Allow the code to access a FileDescriptor, and thus, to read the associated file. So, Confidentiality is no longer ensured. Threat: Espionage.	Granting this permission allows an attacker to read your files or other communication channel like your internet connections.	A typical exploit strategy would be to read the content of a file via its file descriptor.
exitVM	Allow the code to halt the JVM. So, Availability is no longer ensured.	Granting this permission allows a malicious code to halt all the Java	An exploit strategy would be to do DoS on the user's system by halting

	Threat: DoS.	applications currently running on your system.	the JVM, thus terminating the Java programs currently running on the system.
shutdownHooks	Allow the code to control the shutting down of the JVM. So, Availability is no longer ensured. Threat: DoS.	Granting this permission may allow a hacker to stop the execution of the application and provoke inconsistence behaviour	Not implemented.
modifyThreadGroup	Allow the code to add or remove existing threads inside a group of threads. It also permits to control these threads, for example by modifying their proprieties, by stopping them…etc. So, Availability is no longer ensured. Threat: DoS.	Granting this permission may allow an attacker to control the Java programs that are running on your computer, for instance by halting them.	Not implemented.
stopThread	Allow the code to stop a thread of a java program. So, Availability is no longer ensured. Threat: DoS. Note: Thread.stop() is deprecated because it is unsafe.	Granting this permission may allow an attacker to stop the Java programs that are running on your computer, or at least disrupt their behaviour.	A typical exploit strategy would be to stop one thread of a java application in order to crash it. Indeed, they are often many threads running for a single application. They are closely bound: if one crash or do not complete its job, the entire program will, at least, misbehave.
modifyThread	Allow the code to interrupt and resume a thread of a Java program. So, Availability is no longer ensured. Threat: DoS. Note: Thread.suspend() is deprecated because it is unsafe (risk of deadlocks among others).	Granting this permission may allow an attacker to stop temporally the Java programs that are running on your computer.	Not implemented.
getProtectionDomain	Allow the code to obtain the current ProtectionDomain. Using the	Granting this permission allows an attacker to know your security	Not implemented.

	getPermissions() method, the code obtains information about the security policy for various code sources. This information may be used to help the hacker in a future attack. So, Confidentiality is no longer ensured. Threat: Espionage.	configuration and to use that to prepare an attack in a better way.	

java.security.SecurityPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
insertProvider.{provider name}	A malicious code can add a provider (at runtime not permanently). A provider represents trustworthy security algorithms and key generation. So, the SystemManager won't check the authenticity of the future algorithm. So, Integrity is no longer ensured. Threats: fool users.	Granting this permission compromise secure connection.	Not implemented
removeProvider.{provider name}	A malicious code can remove a trustworthy provider thus attack the availability of the application because it could not encrypt or decrypt cipher with the good provider. So the availability is no longer	If you grant this permission, your application can change the way to encrypt your data. Thus, the content of your messages would be inconsistent.	Not implemented

	ensured if the software wants to use the removed provider Threads: DoS.		
putProviderProperty.{provider name}	A malicious code can set other parameters (key, properties) of an existing provider. It is an Integrity attack. Threat: DoS.	If you grant this permission, your application can be able to change the way to encrypt your data.	Exploit strategy: You can add temporally additional information to a provider. This information can be use to cipher in another way. The permission blocks the call of the method setProperty(string s, string t) of the class Provider.
removeProviderProperty.{provider name}	A malicious code can remove parameters (key, name) of an existing provider. So, the availability is no longer ensured. Threat : DoS.	If you grant this permission, your application can be able to change the way to encrypt your data.	Not implemented.
clearProviderProperties.{provider name}	A malicious code can remove the properties of a provider add previously during the runtime. So, the availability is no longer ensured. Threat: DoS.	If you grant this permission, your application can be able to change the way to encrypt your data.	Some softwares place at runtime some additional information about a given Provider. These information can be cleared by the method clear() of the class Provider.
setIdentityInfo	Allow the code to modify the information stored about an Identity (person or company). A malicious code may corrupt this information to deny its access or to fool users. So, Integrity and Availability are no longer ensured. Threats: fool users.	Granting this permission allows the code to change the information which describes the contacts you do business with. This could cause trouble in your future trades.	Not implemented
setProperty.{key}	A malicious code can set a security	Granting this permission allows the	Not implemented

	property. It works only at runtime. It is not a persistent change. Applications can be fooled if the security properties change. So, Integrity is no longer ensured. Threats: fool users.	code to change security properties.	
getPolicy	This permission allows the software to know the policy of the security manager. This permission is not dangerous If there are no other permissions granted because malicious codes will not be able to send it (SocketPermission) or write it on a file (FilePermission) for instance. However it still can help hacker to find flaws in the security policy. So, Confidentiality is not granted anymore. Threat: espionage.	Granting this permission allows the application to know if it can do something bad on your computer. This could help a hacker to prepare an attack against your system.	It shows the list of the permissions granted in the current policy.
createAccessControlContext	Allow the code to create an AccessControlContext. Using the associated DomainCombiner and ProtectionDomain, the code will be able to get the permissions associated with the current ProtectionDomains in the various Threads running. A malicious code may use this information to optimize an attack or to collect them (if SocketPermission is also granted).	Granting this permission allows the code to obtain information about the current security configuration. This could help a hacker to prepare an attack against your system.	Not implemented

	So, Confidentiality is no longer ensured. Threat: espionage.		
getDomainCombiner	Allow the code to access the current DomainCombiner and ProtectionDomain. A malicious code may read the permissions granted for the context. So, Confidentiality is no longer ensured. Threat: espionage.	Granting this permission allows the code to obtain information about the current security configuration. This could help a hacker to prepare an attack against your system.	Not implemented.

java.io.FilePermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
read	Allow to read file. If the read file contains important information, the malicious code can get it. This permission protects the confidentiality. So, Confidentiality is no longer ensured. Threat: espionage.	Granting this permission allows the code to read your documents and possibly find some confidential information	The exploit code reads the five first lines of a given file.

java.net.SocketPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
listen	Allow the code to create a socket and wait for connections from distant hosts. This permission is not sufficient to accept this connection (cf. accept). This permission is necessary on server-side but not sufficient to settle a client-server mechanism. Alone, this permission cannot be used by a hacker to pirate the user's computer.	Granting this permission allows the code to wait for a connection from a distant computer (but not to accept it). There is no direct danger to grant this permission, except if you also grants the "accept" permission.	The exploit code shows a server-side which creates a ServerSocket and waits for a connection. If no connection comes from a client, the execution runs without any problem. If a client connects on the given port, a security exception is launched because the "listen" permission only gives the permission to wait for a connection, not to accept it.
resolve	Enable the use of DNS service. This permission is implied when using the other SocketPermission. It is dangerous only if the malicious code can also force the utilisation of the hacker's DNS server.	Granting this permission enables the code to obtain the network address of your computer in the Internet. There is no danger to grant this permission.	The implemented code uses the DNS service to convert a hostname into an IP.

java.io.SerializablePermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
enableSubclassImplementation	Allow the code to override the default implementation of the serialization and deserialization of object. A malicious code may corrupt the data during serialization or deserialization, access confidential data or simply remove the content of the object. It may be very harmful to the user/company. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: espionage and fool users.	Granting this permission allows the code to modify the way data is processed before its storage (Note: in a serialized form) or sending (Note: in a serialized form) on a communication channel (e.g. on your Internet connections). It is dangerous because your data can be lost or corrupted.	An exploit strategy would be to override the writeObject and readObject methods (cf. ObjectOutputStream and ObjectInputStream) using writeObjectOverride and readObjectOverride respectively. These latter methods will be called instead of the original ones. Inside, an attacker can modified the data before serializating/deserializating it. For instance the hacker may add or shifted bytes of the given object to write/read, thus corrupting it.
enableSubstitution	Allow the code to substitute the object to serialize/deserialize by another. The serialization/deserialization process used is the default one; only the replaceObject and resolveObject methods are re-implemented. A malicious code may corrupt the data during serialization or deserialization, access confidential data or simply remove the content of the object. It may be very harmful to the user/company.	Granting this permission allows the code to substitute data by fake or dangerous one before its storage (Note: in a serialized form) or sending (Note: in a serialized form) on a communication channel (e.g. on your Internet connections). It is dangerous because your data can be lost or corrupted.	The exploit code shows how to replace each String object to serialize by a fake other one. First, the code serializes the string “HELLO WORLD !”. During the serialization process (the default one), the string is replaced by ‘HACKED during serialization!!’. Then, the code deserializes the object and reads it to prove that it is this last value that was serialized.

	So, Confidentiality, Integrity and Availability are no longer ensured. Threats: espionage and fool users.		

javax.net.ssl.SSLPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
getSSLSessionContext	Allow the code to access the context of an SSL session. A session may gather several secure connections. Given the session, a malicious code can obtain information about the various connections (id, cipher type…), invalidate the session, change the timeouts…etc. So, Confidentiality, Integrity and Availability are no longer ensured. Threats: DoS.	Granting this permission allows the code to control the context of a secure connection you opened. It is dangerous because It may make the connection inconsistent or untrustworthy.	The exploit code retrieves the SSLSessionContext on a connection opened via a SSLSocket.

java.sql.SQLPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
setLog (server side)	Allow the call of *javax.sql.DataSource .setLogWriter, javax.sql. ConnectionPoolDataSource. setLogWriter,* and *javax.sql.XADataSource .setLogWriter javax.sql.* These functions are used in servlets to log database connections. On the server side logging allows to save passwords or usernames. This is a hindrance to data confidentiality.	Allow this permission permits to show connections logs of all the clients of your database. It can also show their login/password.	An exploit strategy would be to develop a servlet that uses a database connection and then enables the logs. Finally, the servlet would display all the connection traces. The login/ password used to connect to this database may be in the logs.

java.net.NetPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
setDefaultAuthenticator	Allow the code to customize the mechanism that handles the opening of an URL which is protected by a login/password (.htaccess). A malicious code may create an authentication mechanism that keeps trace of the login/password entered by users. Then, this information may be used to access confidential data or to damage the user's system. However, without other permissions, like FilePermission or SocketPermission, the login/password collected won't be easily retrieved by the hacker. So, Confidentiality and Integrity are no longer ensured. Threats: DoS and espionage.	Granting this permission gives the application the ability to retrieve the password you write in a form.	The exploit code creates a new Authenticator and sets it as the default authenticator. This new authenticator displays the content of an URL protected by a login/password mechanism (.htaccess). It implements its own way to ask for the login/password and processes them (getPasswordAuthentication). The code may modify or store this information. However, this exploit code directly returns the login/password as a PasswordAuthentication object.
requestPasswordAuthentication	Allow the code to ask the user for his password. Because the code handles this procedure, it may steal this information by storing or sending it after (require FilePermission or	Granting this permission allows the code to ask you your password and keep it.	Not Implemented.

	SocketPermission). So, Confidentiality is no longer ensured. Threats: espionage.		

java.awt.AWTPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
createRobot	Allow the code to take control of the graphics events on whole the system. A malicious code may be able to artificially move the mouse and use the keyboard. It can also stop the user from using these devices. Finally, a robot may read the screen and copy it as a BufferedImage (screenshot). So, Confidentiality and Availability are not ensured anymore. Threats: espionage and DoS.	Granting this permission allows the code to control the graphics events (mouse, keyboard, and screen) on the entire system. This is very dangerous because a malicious code can use that to control your computer.	The exploit code moves the mouse on the screen following a square.
accessEventQueue	Allow the code to take control of the graphics events in the application. A malicious code may access the events currently under process and modify or delete them. The resulting behaviour of the application would be inconsistent. So, Confidentiality, Integrity and	Granting this permission allows the code to control your mouse and keyboard in Java applications. Then, they may misbehave.	The exploit code adds an ActionEvent (like a click on a button) to the AWTEventQueue. So, the application will consider the user has really clicked on a button.

	Availability are not ensured anymore. Threats: espionage and DoS.		

Low severity

java.lang.RuntimePermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
queuePrintJob	Allow the code to print, by adding new jobs in the printer's queue. Hackers can use that to waste paper, print confidential data or simply deny the user access to the printer.	This permission allows the code to use your printer. A hacker would be able to overload your printer by printing thousands of papers.	The code put 100 pages in the printing queue, resulting in using the printer's resources and making the users waiting for availability.

java.security.SecurityPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
getProperty.{key}	Allow the code to access the value of the security properties of the Java environment. A malicious code may not be able to affect directly the behaviour of the application but may use this information to optimize the attack. Thus, other permissions are required in order to do real	Granting this permission allows the code to obtain information about the security configuration of your computer. This permission is not directly a danger for the security of your system.	The exploit code displays the security property 'login. configuration.provider' written in the file "java.security".

	damage. So, Confidentiality is not ensured anymore. Threat: espionage.		
printIdentity	Allow the code to display information about a given Identity (person or company): its name, if it is trust or not…etc. A malicious may retrieve confidential information about the Identity objects stored. Without other permissions (Socket, File…) this permission is not very dangerous. So, Confidentiality is no longer ensured. Threat: espionage.	Granting this permission allows the code to obtain the information which describes the contacts you do business with. Granting this permission is not directly dangerous, except if the code can also connect to distant computer.	Not implemented

java.awt.AWTPermission

showWindowWithoutWarningBanner	In applet, a warning banner always tells the user that the window was created by an applet and not by an application. It may be important for a user because applications and applets have not the same default rights. This permission allows the code to not to display this banner. Using this trick, a malicious code may fool users. So, Integrity is not ensured anymore. Threats: fool users.	Granting this permission allows the code to pass an applet window off as an application window. Hackers can use this trick to fool you.	Not implemented.

javax.sound.sampled.AudioPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
record	Allow to record sounds from a computer, thus, permitting eavesdropping. When you record audio, the system use mixers. If many mixers are opened, the computer can become slow, because mixers use lots of resources. It is even more dangerous with file or socket permissions. So, Confidentiality and the	Granting this permission allows the code to record sounds. You can be eavesdropped and the performance of your computer can be affected.	A possible way to exploit this permission is for eavesdropping user's conversations.

	Availability are no longer ensured. Threats: espionage and DoS		
play	This permission allows to play audio on the computer. When you play audio, the system uses mixers. If many mixers are opened, the computer can become slow, because they use lots of resources The availability is no longer ensured. Threat: DoS.	Granting this permission allows the code to play sounds. The performances of your computer can be affected.	The exploit code launches many threads that play the same sound. Thus, all these threads overload the user's cpu.

java.sql.SQLPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
setLog (Client side)	Allow to call the function DriverManager.setLogWriter that traces (in a file or in the console) connections and queries to the database. It does not show any passwords or usernames.	Allow this permission permits to save your database connection logs.	The code shows the logs when it connects to a MySQL database.

java.util.PropertyPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
write	Allow the code to set the properties of the system, for example the OS version and architecture, the user directory, the endian type, the Java version or the file encoding. A malicious code may change the value of these properties to make the system unstable or to modify the initial behaviour of the application. So, Confidentiality and Integrity are not ensured anymore. Threats: fool users and DoS.	Granting this permission allows the code to modify some configuration information of your computer and thus affect the execution of the application.	The exploit code shows a few property values (we also granted the read permission for the demonstration) and then modify the value of the property “user.language” from “fr” to “en”.
read	Allow the code to read the value of properties of the system. A malicious code may use this information to prepare or optimize its damage on the computer, or simply to spy the characteristics of the system. So, Confidentiality is not ensured anymore.	Granting this permission allows the code to obtain information about the configuration of your computer. A hacker may use this information to prepare an attack against your system.	The exploit code shows a few property values, including the OS name and the user directory.

java.util.logging.LoggingPermission

Action	Severity	User-friendly hint	Exploit code comment or Exploit Strategy
control	Allow the code to take control of the logging configuration. A malicious code may be able to remove some logging mechanisms in order to hide what it has done. It can also access and modify the existing mechanisms and add new ones. So, Confidentiality, Integrity and Availability are not ensured anymore. Threats: espionage and fool administrator.	Granting this permission allows the code to log/trace what you are doing in the application.	The exploit code creates a logging mechanism (Logger and ConsoleHandler) with the level INFO. So, only messages with the levels SEVERE, WARNING and INFO will be logged. The code first shows that SEVERE and INFO messages are really displayed and FINE are not. Then, the code modifies the level of the logger to SEVERE. Thus, INFO messages are not logged anymore. The code “removed” information which could have been useful for tracking issues.