This code showcases a classic example of a deserialization vulnerability, which can lead to arbitrary code execution. Let's break down the security risks:

1. **Arbitrary Code Execution**:

   - The class `EvilGadget` implements `Serializable`, indicating it can be serialized and deserialized.

   - Inside the `readObject` method, which is automatically invoked during deserialization, there's a line `Runtime.getRuntime().exec(command);`. This line executes the command stored in the `command` attribute of the object.

   - When an attacker can control the content of the serialized `EvilGadget` object, they can cause arbitrary code to be executed upon its deserialization.

2. **Insecure Serialization**:

   - The `main` method demonstrates the serialization of an `EvilGadget` object with a command of `shutdown`. The serialized object is then base64 encoded and printed.

   - If this serialized data gets into the hands of an attacker, they can potentially reverse the process, modify the command, re-serialize the object, and then feed it to a system that deserializes this object, leading to the execution of the attacker's command.

3. **Duplication**:

   - The class `EvilGadget` appears twice in the provided code. This is not a security risk per se, but it's a redundancy and a potential point of confusion.

How to Mitigate:

1. **Avoid Deserializing Data from Untrusted Sources**: If you must deserialize, make sure you're dealing with trusted data. This is the primary way to prevent deserialization attacks.

2. **Sanitize Input**: If there's any chance that the serialized data might be tampered with or originate from an untrusted source, sanitize and validate the input before deserialization.

3. **Restrict or Remove Dangerous Classes**: Java, for example, allows the creation of a "look-up" filter that restricts which classes can be deserialized.

4. **Update Libraries**: If you're using libraries for serialization/deserialization, ensure they are up-to-date. Some older libraries have known vulnerabilities related to deserialization.

5. **Avoid Custom Deserialization Logic**: As demonstrated by the `readObject` method, custom deserialization logic can be risky. Avoid implementing custom logic unless absolutely necessary, and if you do, ensure it doesn't introduce security risks.

This code is a textbook example of why insecure deserialization can be so dangerous, especially in languages like Java. Always be cautious when working with serialization and ensure you're following best security practices.