Chapter 22 - Supplemental Material: doPrivileged()

Some Java programs run in an environment where the user does not have full control over the program. In other words, the program runs a privileged action on behalf of the user. The idea for the developer is straight-forward:

- I have a privileged action I need run for a user
- I need to verify the user has the proper permission before running the action
- I need to make sure they are limited in what actions they can run
- I need to make sure they don't using caching or other tricks to skip the permission check

1. The doPrivileged() method

A common example Oracle likes to use is reading a system property. The idea is that the programmer is only allowed to read a specific *predefined* system property.

```
import java.security.*;
public class MySecretReader {
   private static final String KEY = "secret.option";
   public String getSecret() {
      return AccessController.doPrivileged(
         new PrivilegedAction<String>() {
            public String run() {
               return System.getProperty(KEY);
            }
         });
   }
}
```

Here `KEY` is a constant that cannot be changed by the user. The idea here is that a user's privilege is temporarily elevated so they can read the `secret.option` value within the system.

2. Ensure Principle of Least Privilege

When executing a privileged action, it is important to ensure that only the minimum access is granted. This is known as the *principle of least privilege*. Can you spot what's wrong with the following example?

```
import java.security.*;
public class MySecretReader {
   public String getSecret(String magicWord) {
      return AccessController.doPrivileged(
         new PrivilegedAction<String>() {
            public String run() {
               return System.getProperty(magicWord);  // DON'T DO THIS!
            }
         });
   }
}
```

In this example, the caller is able to specify which value they want to read. *This is considered a poor practice* as it allows them to read any property within the system. Oracle refers this as a tainted input. Put simply, don't trust anything the user provides when dealing with security. Also use a constant or predefined list to confirm they are accessing only what the original developer intended.

3. Don’t Expose Sensitive Information

Another important aspect of using `doPrivileged()` is ensuring sensitive data is protected. For example, can you spot the security risk in this code?

```
import java.security.*;
import java.util.*;
public class MySecretReader {
   private final List<Integer> codes = ...
   public List<Integer> getSecret() {
      return AccessController.doPrivileged(
         new PrivilegedAction<List<Integer>>() {
            public List<Integer> run() {
               return codes;  // DON'T DO THIS!
            }
         });
   }
}
```

Even though `codes` is marked `final`, the content can still be modified after the `doPrivileged()` is complete. This poses an unacceptable security risk. A much safer version would be to return an immutable copy of the list, such as:

```
return AccessController.doPrivileged(
   new PrivilegedAction<List<Integer>>() {
      public List<Integer> run() {
         return Collections.unmodifiableList(codes);
      }
   }
);
```

4. Don’t Elevate Permissions

Privilege elevation or escalation occurs when a user is mistakenly given access to a higher privilege than they should have access to. One way to prevent privilege elevation is to use to the `AccessController.checkPermission()` method before calling `doPrivileged()`, then execute the command with limited permissions.

```
import java.security.*;
public class MySecretReader {
   public void readData(Runnable task, String path) {
      // Check permission
      Permission permission = new java.io.FilePermission(path,"read");
      AccessController.checkPermission(permission);

      // Execute task with limited permission
      final var permissions = permission.newPermissionCollection();
      permissions.add(permission);
      AccessController.doPrivileged(
         new PrivilegedAction<Void>() {
            public Void run() {
               task.run();
               return null;
            }
         },
         // Using a limited context prevents privilege elevation
         new AccessControlContext(
            new ProtectionDomain[] {
               new ProtectionDomain(null, permissions)
```

```
            })
         );
   }
}
```

5. Be wary of Permission Caching

The last rule you need to know for the exam is to be weary of cached permissions. It is perfectly acceptable to cache permission information, but the permission needs to be checked every time the user accesses it.

For example, assuming there's a `User` class with appropriate attributes, methods and constructors, can you spot the problem in this code?

```
import java.security.*;
import java.util.*;
public class SecretFile {
   private static Map<String, User> data = new HashMap<>();
   public static SecretFile get(String key) {
      var cacheRecord = data.get(key);
      if (cacheRecord != null) {
         // DON'T DO THIS!
         return cacheRecord.getValue();
      }

      final var permission = Permission permission
         = new PropertyPermission(key,"read");
      AccessController.checkPermission(permission);

      final var permissions = permission.newPermissionCollection();
      permissions.add(permission);
      var secret = AccessController.doPrivileged(
         new PrivilegedAction<SecretFile>() {
            public SecretFile run() {
               return new SecretFile();
            }
         }, new AccessControlContext(new ProtectionDomain[] {
               new ProtectionDomain(null, permissions) }));
      data.put(key, new User(secret, permission));
      return secret;
   }
}
```

Did you spot it? It might be hard to see, but there's no permission check when the data is read from the cache! The permission is checked when the data is first read from the cache but not on subsequent calls. We can easily fix this though by checking the permission when it is read from the cache:

```
var cacheRecord = data.get(key);
if (cacheRecord != null) {
   AccessController.checkPermission(cacheRecord.getPermission());
   return cacheRecord.getValue();
}
```

In this example, we see that cached permissions can be safe to use but we have to make sure the permission is validated when it is read the first time and on each request from the cache.

Conclusion

There, you're done! This post covered the overall topics around Privileged Access that you need to know for the 1Z0-819 Exam. The following bullet points summarize the kinds of things you should be watching for on the 1Z0-819 Exam:

- Always validate user input and never allow it to grant access to arbitrary data
- Never give the user unlimited access to the system
- Prevent privilege elevation by validating security
- Never return privileged objects directly or in a way that they can be modified
- Ensure cached permissions are validated on every call