Case Study: Source Code Analysis of a Password Authentication Program to Update a Log File

Dr. Natarajan Meghanathan
CSC 438/539 Systems and Software Security, Spring 2014
Jackson State University
E-mail: natarajan.meghanathan@jsums.edu

We will conduct two case studies: the first case study on a password authentication program and the second case study on a file writer program. The password authentication program is supposed to work as follows: It lets a user to update the log file, provided the user knows the correct password. In this context, the program receives two inputs from the user – the name of the log file to update and the password. If the password entered by the user matches with that stored in the passwordFile.txt file, then the user is allowed to update the log file; otherwise, not. It is assumed the password is one word string. The file writer program is supposed to work as follows: The user inputs the name of the text file to write to. The program then asks the user to input text line by line (a total of 5 lines). Every line input by the user is written to the text file as it is before the next line is input.

We will start with a basic Java program code for each of the above two case studies to accomplish what is required. The programs will be developed using the Java SDK (version 1.7). We will run through these programs in the HP Fortify Source Code Analyzer (SCA) suite to learn about the vulnerabilities present in these code. We will refine the code iteratively to remove one vulnerability at a time. The objective is to refine the code such that the only vulnerability left at the end would be the Poor Logging Practice vulnerability (due to the user of print statements) and the J2EE Bad Practices vulnerability due to the use of the String args[ ] part of the parameters of the main function and with the use of System.exit( ).

We will analyze the following vulnerabilities/weaknesses

- Structural issues (caused due to the layout of the program and how it is written):
  1. System information leak vulnerability
  2. Value never read – Poor programming style
  3. Empty password initialization vulnerability
  4. Hardcoded password vulnerability
- Dataflow issues (caused due to the data input or generated during the program):
  1. Path manipulation vulnerability
- Control flow issues (caused due to the control flow of the program at run-time):
  1. Unreleased resource stream vulnerability
- Semantic issues (caused due to the variations in the context – reading a line, word, etc, at which the program is run)
  1. Denial of service vulnerability

Cast Study # 1: Original Password Authentication Program

```
import java.io.*;
import java.util.*;

class passwdAuth
{

      public static void main(String args[])
      {
            try
            {
                  String logFileName = args[0];
                  File f = new File(logFileName);
                  boolean access_granted = false;
                  String password = "";
                  int integer = 5;

                  if (args.length == 2)
                  {
                        System.out.println("Checking command-line password");
                        password = password + args[1];
                        if (password.equals("3dTAqb.7"))
                        {
                              access_granted = true;
                              System.out.println("Password matches.");
                        }
                        else
                              System.out.println("Command-line password does
not match");
                  }//end if

                  if (access_granted)
                  {
                        System.out.println("Access granted!");
                        PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
                        out.println();
                        out.print("Updated...");
                        out.println();
                        out.flush();
                        out.close();
                  }//end if


            }//end try
            catch (Exception e)
            {
                  System.out.println("an error has occured.");
                  e.printStackTrace();
            }
      }//end main

}//end class
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan passwdAuth.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[ED9B7EF373B01A339705986CF1046949 : medium : System Information Leak : semantic
]
passwdAuth.java(46) : Throwable.printStackTrace()

[5E5378A1512F23833009849E4998C980 : medium : Path Manipulation : dataflow ]
passwdAuth.java(12) :  ->new File(0)
    passwdAuth.java(11) : <=> (logFileName)
    passwdAuth.java(7) :  ->passwdAuth.main(0)

[CD62823DB18FA43912256BE73D261DDA : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    passwdAuth.java(7)

[FD7101EE9C331AF720D73EEC6FB37E6A : high : Password Management : Empty Password
: structural ]
    passwdAuth.java(14)
    Variable: password [passwdAuth.java(14)]

[B13C00179491317A5783F445FC10B6F1 : low : Poor Style : Value Never Read : struct
ural ]
    passwdAuth.java(15)
    Variable: integer [passwdAuth.java(15)]

[440A9BE89C84CA97CF0530B6CBB07250 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth.java(19)

[42A8F7980BCC4E1DD64DA6E1EA515C76 : high : Password Management : Hardcoded Passw
ord : structural ]
    passwdAuth.java(21)

[440A9BE89C84CA97CF0530B6CBB07251 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth.java(24)

[440A9BE89C84CA97CF0530B6CBB07252 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth.java(27)

[440A9BE89C84CA97CF0530B6CBB07253 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth.java(32)

[440A9BE89C84CA97CF0530B6CBB07254 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
```

1.1 System Information Leak Vulnerability

The System Information Leak vulnerability (structural issue) arises when the print statements, in response to unexpected inputs or execution flow, can be used to infer critical information about the program structure, including the sequence of method calls (the call stack). Developers include print statements to facilitate debugging for any erroneous behavior/input. Often such print statements are left in the code even after deployment. Attackers could take advantage of this vulnerability and pass cleverly crafted suitable input values to the program so that it generates the informative error messages. For example, there is a difference between displaying a generic error message like 'Incorrect username or password' compared to a more informative error message like 'Access denied'. In the former case, one cannot easily infer whether the username is wrong or the password is wrong, essentially perplexing the attacker whether or not there exists an account with the particular username passed as input. However, with the 'Access denied' error message, the attacker could easily infer that there exists an account with the attempted username; it is only the password that is invalid. Nevertheless, generic error messages baffle the genuine users of the system and would not be able to infer much from these messages. Thus, there is a tradeoff between the amount of information displayed through the error messages and the ease associated with identifying or debugging the error and fixing the problem.

```
import java.io.*;
import java.util.*;

/* Removing the system information leak vulnerability by printing only
generic (non-detailed) error messages */

class passwdAuth
{

      public static void main(String args[])
      {
            try
            {
                  String logFileName = args[0];
                  File f = new File(logFileName);
                  boolean access_granted = false;
                  String password = "";
                  int integer = 5;

                  if (args.length == 2)
                  {
                        System.out.println("Checking command-line password");
                        password = password + args[1];
                        if (password.equals("3dTAqb.7"))
                        {
                              access_granted = true;
                              System.out.println("Password matches.");
                        }
                        else
                              System.out.println("Command-line password does
not match");
                  }//end if

                  if (access_granted)
                  {
                        System.out.println("Access granted!");
```

```
                        PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
                        out.println();
                        out.print("Updated...");
                        out.println();
                        out.flush();
                        out.close();
                  }//end if


            }//end try
            catch (Exception e)
            {
                  System.out.println("an error has occured.");
                  //e.printStackTrace();
            }
      }//end main

}//end class
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan passwdAuth_1.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[5E5378A1512F23833009849E4998C980 : medium : Path Manipulation : dataflow ]
passwdAuth_1.java(12) :  ->new File(0)
    passwdAuth_1.java(11) : <=> (logFileName)
    passwdAuth_1.java(7) :  ->passwdAuth.main(0)

[CD62823DB18FA43912256BE73D261DDA : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    passwdAuth_1.java(7)

[FD7101EE9C331AF720D73EEC6FB37E6A : high : Password Management : Empty Password
: structural ]
    passwdAuth_1.java(14)
    Variable: password [passwdAuth_1.java(14)]

[B13C00179491317A5783F445FC10B6F1 : low : Poor Style : Value Never Read : struct
ural ]
    passwdAuth_1.java(15)
    Variable: integer [passwdAuth_1.java(15)]

[440A9BE89C84CA97CF0530B6CBB07250 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_1.java(19)

[42A8F7980BCC4E1DD64DA6E1EA515C76 : high : Password Management : Hardcoded Passw
ord : structural ]
    passwdAuth_1.java(21)

[440A9BE89C84CA97CF0530B6CBB07251 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_1.java(24)

[440A9BE89C84CA97CF0530B6CBB07252 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_1.java(27)

[440A9BE89C84CA97CF0530B6CBB07253 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_1.java(32)

[440A9BE89C84CA97CF0530B6CBB07254 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_1.java(45)
```

1.2 Value Never Read – Poor Programming Style

It is not at all a good programming practice to declare variables and never use them(structural issue). We will get rid off the integer (initialized to 5) variable in our authentication program.

```
import java.io.*;
import java.util.*;
/* Handling the "Value never read" Poor style warning */

class passwdAuth
{
	public static void main(String args[])
	{
		try
		{
			String logFileName = args[0];
			File f = new File(logFileName);
			boolean access_granted = false;
			String password = "";
			//int integer = 5;

			if (args.length == 2)
			{
				System.out.println("Checking command-line password");
				password = password + args[1];
				if (password.equals("3dTAqb.7"))
				{
					access_granted = true;
					System.out.println("Password matches.");
				}
				else
					System.out.println("Command-line password does
not match");
			}//end if

			if (access_granted)
			{
				System.out.println("Access granted!");
				PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
				out.println();
				out.print("Updated...");
				out.println();
				out.flush();
				out.close();
			}//end if


		}//end try
		catch (Exception e)
		{
			System.out.println("an error has occured.");
		}
	}//end main

}//end class
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>javac passwdAuth_2.java

C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan passwdAuth_2.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[5E5378A1512F23833009849E4998C980 : medium : Path Manipulation : dataflow ]
passwdAuth_2.java(14) :  ->new File(0)
    passwdAuth_2.java(13) : <=> (logFileName)
    passwdAuth_2.java(9) :  ->passwdAuth.main(0)

[CD62823DB18FA43912256BE73D261DDA : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    passwdAuth_2.java(9)

[FD7101EE9C331AF720D73EEC6FB37E6A : high : Password Management : Empty Password
: structural ]
    passwdAuth_2.java(16)
    Variable: password [passwdAuth_2.java(16)]

[440A9BE89C84CA97CF0530B6CBB07250 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_2.java(21)

[42A8F7980BCC4E1DD64DA6E1EA515C76 : high : Password Management : Hardcoded Passw
ord : structural ]
    passwdAuth_2.java(23)

[440A9BE89C84CA97CF0530B6CBB07251 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_2.java(26)

[440A9BE89C84CA97CF0530B6CBB07252 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_2.java(29)

[440A9BE89C84CA97CF0530B6CBB07253 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_2.java(34)

[440A9BE89C84CA97CF0530B6CBB07254 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_2.java(47)
```

1.3 Empty Password Initialization Vulnerability

The Empty Password vulnerability (structural issue) is associated with the common practice of programmers to initialize variables whose appropriate values are either input by the user or dynamically set depending on the flow of execution of the program at run-time. Developers typically include such variables to validate whether users had entered the appropriate value (during input operations) or check if the desired flow of execution occurred for the program at run-time. The vulnerability associated with such a programming approach is that if an attacker gets to know the invalid value initialized for a critical variable (like the password String in our file writer program), and manages to pass input matching to the invalid value or alter the flow of execution such that the value of the variable remains unchanged, then the program could be potentially under risk of being executed with the invalid value. For example, an object could be initialized to 'null' with the expectation that given the different control paths the program could take, it will point to some valid memory location at run-time; an attacker could manipulate the inputs and/or the control flow of the program such that an operation (say a function call) is invoked on the object when it is still not pointing to any valid memory location. This could terminate the program and typically lead to the familiar *NullPointerException* in Java or the Segmentation Fault in C/C++; at the worst case, the system running the program could crash.

The solution we suggest is not to initialize any critical variable to an invalid value and expect them to be appropriately set by the user or through the execution flow of the program at run-time. As a solution, we suggest combining both variable declaration and value assignment into one single statement instead of having one declaration and multiple statements (though only at most one of them will be executed at run-time depending on the program flow) to assign a possible value. By combining variable declaration and value assignment as one atomic statement, we avoid the situation where the initial invalid value assigned to the variable remains unchanged due to an attacker's manipulations of the control flow of the program.

```
import java.io.*;
import java.util.*;

/* Removing the Empty passwd initialization vulnerability */

class passwdAuth
{

      public static void main(String args[])
      {
            try
            {
                  String logFileName = args[0];
                  File f = new File(logFileName);
                  boolean access_granted = false;

                  if (args.length == 2)
                  {
                        System.out.println("Checking command-line password");
                        String password = args[1];
                        if (password.equals("3dTAqb.7"))
                        {
                              access_granted = true;
                              System.out.println("Password matches.");
                        }
                        else
```

```
                              System.out.println("Command-line password does
not match");
                  }//end if

                  if (access_granted)
                  {
                        System.out.println("Access granted!");
                        PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
                        out.println();
                        out.print("Updated...");
                        out.println();
                        out.flush();
                        out.close();
                  }//end if


            }//end try
            catch (Exception e)
            {
                  System.out.println("an error has occured.");
            }
      }//end main

}//end class
```

```
[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[5E5378A1512F23833009849E4998C980 : medium : Path Manipulation : dataflow ]
passwdAuth_3.java(14) :  ->new File(0)
    passwdAuth_3.java(13) : <=> (logFileName)
    passwdAuth_3.java(9) :  ->passwdAuth.main(0)

[CD62823DB18FA43912256BE73D261DDA : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    passwdAuth_3.java(9)

[440A9BE89C84CA97CF0530B6CBB07250 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_3.java(19)

[42A8F7980BCC4E1DD64DA6E1EA515C76 : high : Password Management : Hardcoded Passw
ord : structural ]
    passwdAuth_3.java(21)

[440A9BE89C84CA97CF0530B6CBB07251 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_3.java(24)

[440A9BE89C84CA97CF0530B6CBB07252 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_3.java(27)

[440A9BE89C84CA97CF0530B6CBB07253 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_3.java(32)

[440A9BE89C84CA97CF0530B6CBB07254 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_3.java(45)
```

1.4 Path Manipulation Vulnerability

The path manipulation vulnerability (data flow issue) arises when the inputs entered by the user are directly embedded into the program and executed, without validating the correctness and appropriateness of the input values. The vulnerability becomes more critical if the user values are directly embedded into path statements that read a critical system resource (say a file) and the program executes them with elevated privileges (higher than that of the user who passed the input). We suggest the approach of sanitizing/filtering the user input by validating it with a blacklist of non-allowable characters and a white list of allowable characters. In this context, we present a sanitizer code in Java that scans the user input for the path/name of the log file with respect to a blacklist of non-allowable characters and a white list of allowable characters. Note that we have moved the code segment to create a file descriptor for the log file to the if block that runs only if the access_granted Boolean variable is set to true. There is no need to unnecessarily create a file descriptor.

```
import java.io.*;
import java.util.*;

/* Removing the Path Manipulation vulnerability */

class passwdAuth
{

      public static int Sanitize(String logFileName){

            if (!logFileName.endsWith(".txt")) return -1;
            // attempting to write to a non-text file

            if (logFileName.indexOf('/') != -1) return -1;
      // attempting to write to a file that is located in another directory

            return 1; // valid file extension; file located in same directory

      }

      public static void main(String args[])
      {
            try
            {
                //  String logFileName = args[0];
                //  File f = new File(logFileName);
                  boolean access_granted = false;

                  if (args.length == 1)
                  {
                        System.out.println("Checking command-line password");
                        String password = args[0];
                        if (password.equals("3dTAqb.7"))
                        {
                              access_granted = true;
                              System.out.println("Password matches.");
                        }
                        else
                              System.out.println("Command-line password does
not match");
                  }//end if
```

```
                if (access_granted)
                {

                    System.out.print("Enter a name for the log file: ");
                    Scanner inputScanner = new Scanner(System.in);
                    String logFileName = inputScanner.next();

                    if (Sanitize(logFileName) == -1){
                       System.out.println("invalid file extension/path");
                       System.exit(1);

                    }

                    File f = new File(logFileName);

                       System.out.println("Access granted!");
                       PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
                       out.println();
                       out.print("Updated...");
                       out.println();
                       out.flush();
                       out.close();
                }//end if


            }//end try
            catch (Exception e)
            {
                  System.out.println("an error has occured.");
            }
      }//end main

}//end class
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan passwdAuth_4.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[A2D8FFFE7B4A7198640C07FA649EADEC : low : J2EE Bad Practices : System.exit : sem
antic ]
passwdAuth_4.java(51) : System.exit()

[CD62823DB18FA43912256BE73D261DDA : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    passwdAuth_4.java(21)

[440A9BE89C84CA97CF0530B6CBB07250 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(31)

[42A8F7980BCC4E1DD64DA6E1EA515C76 : high : Password Management : Hardcoded Passw
ord : structural ]
    passwdAuth_4.java(33)

[440A9BE89C84CA97CF0530B6CBB07251 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(36)

[440A9BE89C84CA97CF0530B6CBB07252 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(39)

[440A9BE89C84CA97CF0530B6CBB07253 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(45)

[440A9BE89C84CA97CF0530B6CBB07254 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(50)

[440A9BE89C84CA97CF0530B6CBB07255 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(57)

[440A9BE89C84CA97CF0530B6CBB07256 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_4.java(70)
```

1.5 Password Management: Hard coded Password Vulnerability

The Hardcoded Password vulnerability (structural issue) is about leaving passwords in plaintext as part of the source code. If an attacker gets access to the code (even in the form of an executable), he can reverse engineer it and get to know the password hardcoded in the program. With the help of this password, the attacker would then be able to easily get access to the critical system resources that were meant to be protected. To fix this vulnerability, we suggest reading the password from a file as part of the program and compare with the user-entered password. This way, unless the attacker gets access to the file in which the password is stored, he cannot easily break-in.

For increased security, we even suggest to store the password in an encrypted form in a file, retrieve through the program and store in the main memory in the encrypted form itself. The user-entered password is encrypted as part of the program code, and the two encrypted ciphertexts could be compared (bit-by-bit) for user authentication. We would even prefer a keyed encryption if the key could be securely stored and retrieved for encryption at the time of authenticating the user.

```
import java.io.*;
import java.util.*;

/* Removing the Hardcoded Password Vulnerability by reading the actual
password from a file*/
/* We use the Scanner class to read the contents of a file */

class passwdAuth
{

      public static int Sanitize(String logFileName){

            if (!logFileName.endsWith(".txt")) return -1;
            // attempting to write to a non-text file

            if (logFileName.indexOf('/') != -1) return -1;
      // attempting to write to a file that is located in another directory

            return 1; // valid file extension; file located in same directory

      }

      public static void main(String args[])
      {
            try
            {
                  boolean access_granted = false;

                  if (args.length == 1)
                  {
                        System.out.println("Checking command-line password");
                        String inputPassword = args[0];

                        File f = new File("passwordFile.txt");
                        Scanner fileScanner = new Scanner(f);
                        String actualPassword = fileScanner.nextLine();

                        if (inputPassword.equals(actualPassword))
                        {
                              access_granted = true;
```

```
                              System.out.println("Password matches.");
                        }
                        else
                              System.out.println("Command-line password does
not match");
                  }//end if

                  if (access_granted)
                  {

                     System.out.print("Enter a name for the log file: ");
                     Scanner inputScanner = new Scanner(System.in);
                     String logFileName = inputScanner.next();

                     if (Sanitize(logFileName) == -1){
                        System.out.println("invalid file extension/path");
                        System.exit(1);

                     }

                     File f = new File(logFileName);

                        System.out.println("Access granted!");
                        PrintWriter out = new PrintWriter(new
FileOutputStream(f, true));
                        out.println();
                        out.print("Updated...");
                        out.println();
                        out.flush();
                        out.close();
                  }//end if


            }//end try
            catch (Exception e)
            {
                  System.out.println("an error has occured.");
            }
      }//end main

}//end class
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan passwdAuth_5.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[A2D8FFFE7B4A7198640C07FA649EADEC : low : J2EE Bad Practices : System.exit : sem
antic ]
passwdAuth_5.java(54) : System.exit()

[CD62823DB18FA43912256BE73D261DDA : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    passwdAuth_5.java(21)

[440A9BE89C84CA97CF0530B6CBB07250 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(29)

[440A9BE89C84CA97CF0530B6CBB07251 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(39)

[440A9BE89C84CA97CF0530B6CBB07252 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(42)

[440A9BE89C84CA97CF0530B6CBB07253 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(48)

[440A9BE89C84CA97CF0530B6CBB07254 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(53)

[440A9BE89C84CA97CF0530B6CBB07255 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(60)

[440A9BE89C84CA97CF0530B6CBB07256 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    passwdAuth_5.java(73)
```

Cast Study # 2: File Writer Program

```
import java.io.*;

class fileWriter {

 public static void main(String[ ] args) throws IOException{

 try{
      FileWriter fw = new FileWriter(args[0]);
      PrintWriter pw = new PrintWriter(fw);
      BufferedReader br = new BufferedReader(new InputStreamReader(
System.in));

      for (int lineNum = 1; lineNum <=5; lineNum++){
            System.out.print("Enter line # "+lineNum+" : ");
            String line = br.readLine( );
            pw.println(line);
      }

      pw.close( );
      fw.close( );

 }
 catch(IOException ie){
      ie.printStackTrace( );
 }
 }

}
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan fileWriter.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[C4CAF7E83C631F366D678FC3044CB76C : low : Denial of Service : semantic ]
fileWriter.java(14) : BufferedReader.readLine()

[01796F93820266EFFCFF2F031B195EB1 : medium : System Information Leak : semantic
]
fileWriter.java(23) : Throwable.printStackTrace()

[AEA394670F3E21F6326BFBF779ACCBFB : medium : Path Manipulation : dataflow ]
fileWriter.java(8) :  ->new FileWriter(0)
    fileWriter.java(5) :  ->fileWriter.main(0)

[F266945834DE5768D8CD8BD724F800B0 : medium : Unreleased Resource : Streams : con
trolflow ]

    fileWriter.java(8) : start -> loaded : fw = new FileWriter(...)
    fileWriter.java(8) : loaded -> loaded : fw refers to an allocated resource
    fileWriter.java(8) : loaded -> loaded : fw refers to an allocated resource
    fileWriter.java(9) : loaded -> loaded : pw = new PrintWriter(fw)
    fileWriter.java(9) : loaded -> loaded : pw refers to an allocated resource
    fileWriter.java(9) : loaded -> loaded : pw refers to an allocated resource
    fileWriter.java(12) : Branch taken: (lineNum <= 5)
    fileWriter.java(14) : java.io.IOException thrown
    fileWriter.java(14) : loaded -> loaded : fw no longer refers to an allocated
 resource
    fileWriter.java(14) : loaded -> loaded : pw no longer refers to an allocated
 resource
    fileWriter.java(14) : loaded -> loaded : fw no longer refers to an allocated
 resource
    fileWriter.java(14) : loaded -> loaded : pw no longer refers to an allocated
 resource
    fileWriter.java(14) : loaded -> end_of_scope : pw end scope : Resource leake
d : java.io.IOException thrown

[75578FE56F1114762BF7A32BC4CB48D9 : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    fileWriter.java(5)

[1A778D1F28FA742D0134E0F8F3C2964F : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter.java(13)
```

2.1 Unreleased Resource Stream Vulnerability

The Unreleased Resource vulnerability (control flow issue) arises when the developer does not include appropriate code that would guarantee the release of the system resources after their intended use (i.e., the resources are no longer needed in the program).

In the context of the file writer program, the unreleased resource vulnerability was attributed due to the possibility of the stream objects (the FileWriter and PrintWriter) that were instantiated to write the lines to the text file not being released due to unanticipated errors during the file print (write) operation; the control flow of the program could immediately switch to the *catch* block for the *IOException* (the exception class handling most of the file read/write errors in Java) and not executing the release statements (the close method calls) for the streams in the *try* block. Once the control exits the *catch* block, the program continues to normally execute the code following the entire *try-catch* block and does not go back to execute the remaining code in the *try* block where the error occurred. Hence, at most care should be taken to release the resources that were allocated inside the *try* block.

However, since there can be more one than *catch* block, for a single *try* block, depending on the specific exceptions that could be generated from the *try* block and need to be caught and processed, it would not be a good idea to include statements pertaining to the release of the allocated system resources in a particular *catch* block. This is because *catch* blocks are supposed to be listed in the increasing order of the scope of the possible exceptions that could be generated from the *try* block, and hence if an exception that is higher up in the hierarchy than an *IOException* is generated before completing the file write operation, then the control would go to a *catch* block that is included somewhere below the *catch* block for the *IOException* class, and the stream resources allocated for the file read operation would not be released at all. To avoid such a scenario, one solution is to include redundant resource release statements in each of the *catch* blocks. This would unnecessarily increase the code size and the memory footprint of the program. An alternate and better solution is to include a *finally* block (optional in Java and need not be used along with a *try-catch* block) following the *catch* blocks and include in it all the statements related to the release of the allocated system resources in the corresponding *try* block. The good news is the Java Virtual Machine will definitely execute the *finally* block after executing one or more *catch* block(s), and will not skip it. This way, the allocated system resources in the *try* block are guaranteed to be released.

Note that, we cannot directly call the *close*( ) method on the file writer object *fw*; it would generate an IOException. Hence, we have to call the close( ) method inside a try-catch block within the finally block. For clarity in the code, we have create a safeClose( ) method (called from the finally block) and close the file writer inside that method.

```
import java.io.*;
import java.util.*;

/* Removing the Unreleased resource stream vulnerability on the resources
FileWriter and PrintWriter objects by closing them in the finally block */
/* Need to handle the IOException generated by the close() methods */

class fileWriter {

 public static void safeClose(FileWriter fw){

 try{
    fw.close();
  }
 catch(IOException e){
```

```
      System.out.println("Error in closing file writer...");
   }

 }


 public static void main(String[ ] args){

      FileWriter fw = null;
      PrintWriter pw = null;

 try{
      fw = new FileWriter(args[0]);
      pw = new PrintWriter(fw);
      BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

      for (int lineNum = 1; lineNum <=5; lineNum++){
            System.out.print("Enter line # "+lineNum+" : ");
            String line = br.readLine( );
            pw.println(line);
      }

      pw.close( );
      fw.close( );

 }
 catch(Exception ie){
      //ie.printStackTrace( );
      System.out.println("Error occurred..");
 }
 finally{

    if (fw != null)
      safeClose(fw);

    if (pw != null)
      pw.close();
   }
 }

}
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>javac fileWriter_1.java

C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan fileWriter_1.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[C4CAF7E83C631F366D678FC3044CB76C : low : Denial of Service : semantic ]
fileWriter_1.java(34) : BufferedReader.readLine()

[AEA394670F3E21F6326BFBF779ACCBFB : medium : Path Manipulation : dataflow ]
fileWriter_1.java(28) :  ->new FileWriter(0)
    fileWriter_1.java(22) :  ->fileWriter.main(0)

[946B4F990D6826E50BB4E59068160B0E : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_1.java(15)

[75578FE56F1114762BF7A32BC4CB48D9 : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    fileWriter_1.java(22)

[1A778D1F28FA742D0134E0F8F3C2964F : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_1.java(33)

[1A778D1F28FA742D0134E0F8F3C29650 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_1.java(44)
```

2.2 Denial of Service Vulnerability

The denial of service vulnerability in the file writer program occurs due to the use of the readLine( ) method of the BufferedReader class. With the readLine( ) method, an attacker can launch a denial of service attack forcing the program to buffer the user inputs until a '\n' character is found. The method will not throw any exception if a '\n' character is not found; it will continue to buffer the input until the newline character is found. We remove the above vulnerability by using the Scanner class and make it to read the input passed by the user using the nextLine( ) method. The nextLine( ) method also buffers the characters until a newline character is read; but the maximum number of characters that can be read is 1024. The SCA does not raise the denial of service vulnerability when the nextLine( ) method of Scanner is used to read a line.

```
import java.io.*;
import java.util.*;

/* Remove the denial of service vulnerability by using the Scanner nextLine(
) method to read the lines input by the user */

class fileWriter {

 public static void safeClose(FileWriter fw){

 try{
    fw.close();
  }
 catch(IOException e){
      System.out.println("Error in closing file writer...");
  }

 }



 public static void main(String[ ] args) throws IOException{

     FileWriter fw = null;
     PrintWriter pw = null;

 try{
      fw = new FileWriter(args[0]);
      pw = new PrintWriter(fw);

      Scanner lineReader = new Scanner(System.in);

      for (int lineNum = 1; lineNum <=5; lineNum++){
            System.out.print("Enter line # "+lineNum+" : ");
            String line = lineReader.nextLine( );
            pw.println(line);
      }

      pw.close( );
      fw.close( );

 }
```

```
 catch(IOException ie){
      //ie.printStackTrace( );
      System.out.println("Error occurred..");
 }
 finally{

    if (fw != null)
      safeClose(fw);

    if (pw != null)
      pw.close();
  }
 }

}
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan fileWriter_2.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[AEA394670F3E21F6326BFBF779ACCBFB : medium : Path Manipulation : dataflow ]
fileWriter_2.java(27) :  ->new FileWriter(0)
    fileWriter_2.java(21) :  ->fileWriter.main(0)

[946B4F990D6826E50BB4E59068160B0E : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_2.java(14)

[75578FE56F1114762BF7A32BC4CB48D9 : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    fileWriter_2.java(21)

[1A778D1F28FA742D0134E0F8F3C2964F : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_2.java(33)

[1A778D1F28FA742D0134E0F8F3C29650 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_2.java(44)
```

2.3 Path Manipulation Vulnerability

In this case study, the path manipulation vulnerability arises due to the direct embedding of the user-supplied input (args[0]) to the File constructor. Instead, we get the name of the file input as a String using the Scanner class next( ) method, and then pass that String as the argument for the File constructor. We could add the Sanitize( ) method of Case Study 1 - Section 1.4 here, if we are interested in making sure the target file is supposed to be a text file and should be located in the same directory as that of the program. To keep it simple, we assume there is no need for sanitization in this particular case study.

```
import java.io.*;
import java.util.*;

class fileWriter {

 public static void safeClose(FileWriter fw){

 try{
    fw.close();
  }
 catch(IOException e){
      System.out.println("Error in closing file writer...");
  }

 }



 public static void main(String[ ] args) throws IOException{

     FileWriter fw = null;
     PrintWriter pw = null;

 try{

       Scanner inputScanner = new Scanner(System.in);
       String fileName = inputScanner.next();

       fw = new FileWriter(fileName);
       pw = new PrintWriter(fw);

       Scanner lineReader = new Scanner(System.in);

       for (int lineNum = 1; lineNum <=5; lineNum++){
             System.out.print("Enter line # "+lineNum+" : ");
             String line = lineReader.nextLine( );
             pw.println(line);
       }

       pw.close( );
       fw.close( );

 }
 catch(IOException ie){
       //ie.printStackTrace( );
```

```
      System.out.println("Error occurred..");
 }
 finally{

    if (fw != null)
      safeClose(fw);

    if (pw != null)
      pw.close();
  }
 }

}
```

```
C:\3300-laptop\JSU-Teaching\Spring-Semesters\Spring2014\CSC438-539\SCAExamples-F
inal>sourceanalyzer -jdk 1.7 -scan fileWriter_3.java


[C:/3300-laptop/JSU-Teaching/Spring-Semesters/Spring2014/CSC438-539/SCAExamples-
Final]

[946B4F990D6826E50BB4E59068160B0E : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_3.java(12)

[75578FE56F1114762BF7A32BC4CB48D9 : low : J2EE Bad Practices : Leftover Debug Co
de : structural ]
    fileWriter_3.java(19)

[1A778D1F28FA742D0134E0F8F3C2964F : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_3.java(36)

[1A778D1F28FA742D0134E0F8F3C29650 : low : Poor Logging Practice : Use of a Syste
m Output Stream : structural ]
    fileWriter_3.java(47)
```