Gjøvik University College

Software Security

Assignement 4

SecureDesktop - Code Audit

Student:
Tommy B. Ingdal

October 8, 2014

Contents

1 Introduction

In this paper I will do an audit of the SecureDesktop application by using static code analysis tools to discover potential vulnerabilities. The application is written in Pascal/Delphi, so the tools of choice is Pascal Analyzer (PAL) and SourceMonitor.
I will (very) briefly describe what each of these tools do, and what information we can learn from them.

1.1 Static Code Analysis

Static Code Analysis [3] or Source Code Analysis referes to the process of running static code analysis tools in an attempt of discovering potential vulnerabilities in an application. Since this method is *static* we don't actually run the application. Instead the analysis tools analyze the code by searching through the source code using techniques such as Taint Analysis and Data Flow Analysis.

Taint Analysis tries to identify data that has been modified (tainted) by the users. User supplied data is traced, and if this data is passed to a vulnerable function it is flagged as a vulnerability.

Data Flow Analysis is used to, dynamically, collect information about data in software, while still being in a static state. There are three common terms used in data flow analysis. You can read more about this on owasp.org.

2 Tools

2.1 Pascal Analyzer

Pascal Analyzer [2], or PAL for short, parses Delphi or Borland Pascal source code. It builds large internal tables of identifiers, and collects other information such as calls between subprograms.

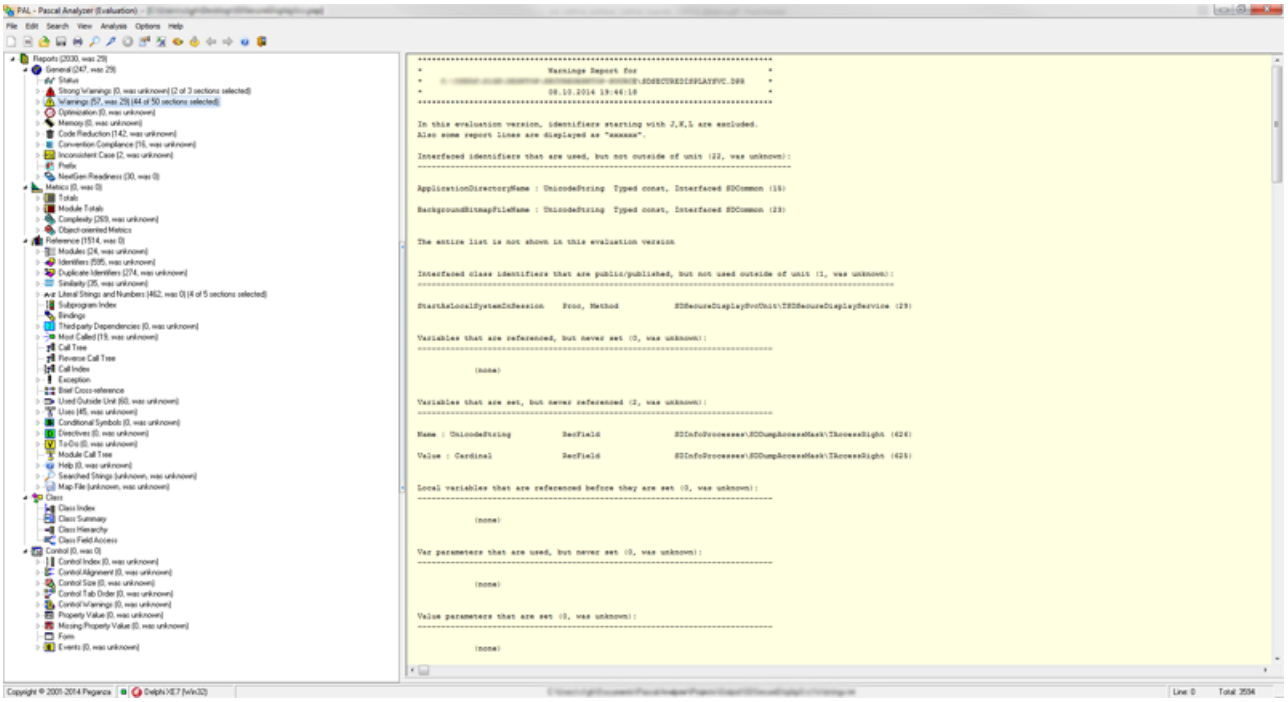

Figure 1: Pascal Analyzer 3.5.

2.2 Source Monitor

SourceMonitor [4] lets you see inside your software source code to find out how much code you have and to identify the relative complexity of your modules.

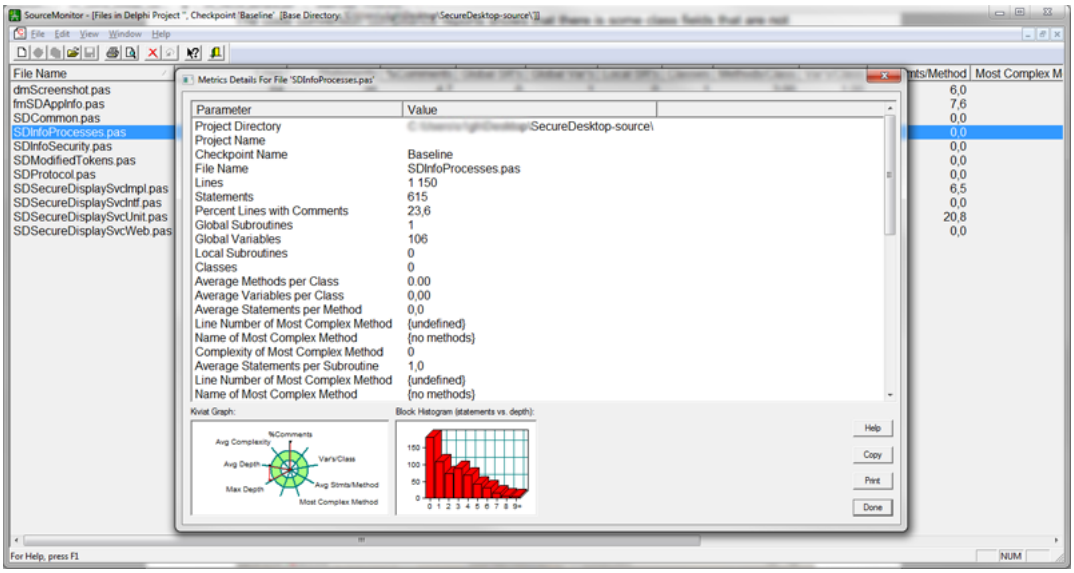

Figure 2: SourceMonitor.

3 Audit

3.1 Review Strategy

Since I have minimal experience with Pascal/Delphi I first had to read about the programming language [1] to be able to understand the code. Also, since I have never used Pascal Analyzer or SourceMonitor before I had to learn these tools as well.

In conjunction with Pascal Analyzer I used SourceMonitor to get a better understanding of the code. SourceMonitor gave me a good overview of all the function names, how many global variables the application uses and also some information of the complexity of the code.
After the static code analysis I had to review the code manually.

3.2 Results

3.2.1 SDCommon.pas

Line 16

```
ConfigFileName = 'SecureDesktop.ini';
```

Since we don't know what this file contains we can't be sure, but what if someone manipulate the data stored in this file?

Line 23

```
BackgroundBitmapFileName = 'Background.bmp';
```

What will happen if we create a huge file (e.g 64GB) and name it Background.bmp? Will the program crash?

Line 118

```
function SDServiceRequestURL: string;
var
  INI: TINIFile;
begin
  INI := TINIFile.Create(Format('%s\%s', [ConfigDirectory,
                                          ConfigFileName]));
  Result := INI.ReadString(INISection_ServiceRequest,
                           INIIdent_RequestURL, '');
  INI.Free;
end;
```

By changing the *INIIdent_RequestURL* can we possibly change the flow of the application? Maybe force the application to contact malicious websites?

Line 320

```
function SDAppInfoLogoFileName(const nApplicationKey: Integer):
                                        string;
begin
  if (nApplicationKey >= 0) then
  begin
    Result := Format('Logo%.4d.png', [nApplicationKey]);
  end
  else
  begin
    Result := 'invalid.png';
  end;
end;
```

This function constructs an image filename, but where is the check to see if the file actually exists? Can we crash the program with a file that does not exist?

3.2.2 SDInfoSecurity.pas

Line 409

```
cbName := 2048 + 1;
GetMem(OwnerName, cbName);
cbReferencedDomainName := 2048 + 1;
GetMem(ReferencedDomainName, cbReferencedDomainName);
```

cbName and *cbReferencedDomainName* has a fixed size of 2048 bytes, but if the author want to change this in the future, the code has to refactored. This is a bad coding habit and should have been solved with a **const** instead.

There is also two calls to the GetMem function which allocates memory on the heap, but the allocated memory is never released. Not a vulnerability per se, but a weakness. You should always release your allocated memory.

3.2.3 General

1) Multiple hardcoded paths exist throughout the code. This is a really bad programming habit and may lead to the program not functioning properly on other systems.
2) The built-in logging functionality is very detailed and should be kept safe. An attacker may target the log file in order to learn more about the system.

3) Lack of documentation makes it difficult to really understand the underlaying functionality of the application. There's also alot of source code that is commented out. These lines should be removed since it makes reading the code harder.
4) Some files are missing and there is no way to compile the application. Black-box testing [6] is difficult and the people doing the code audit may miss important vulnerabilities. For a reliable result/code audit all the files should be present.
5) The application uses alot of global variables which is a minus.

4 Summary

The use of global variables, hard coded paths and static values makes the application unreliable on systems other than the system where the application was developed. The application may or may not work.
There also exist some vulnerabilities that should be fixed before the application is production ready. By re-factoring the code and removing unnecessary code/comments and using e.g Pascal Analyzer as a guideline, the application can be made ready for deployment.

Since there was no way to compile the application we had to rely on static code analysis and black-box testing, which is difficult. The results presented in this paper should be taken with a grain of salt and only used as a guideline when patching the SecureDesktop application [5].

References

[1] Free Pascal. Programmers's Guide. http://www.freepascal.org/docs-html/prog/prog.html. [Online; accessed 08-October-2014].

[2] Peganza. Pascal Analyzer. http://www.peganza.com/products_pal.htm. [Online; accessed 08-October-2014].

[3] The Open Web Application Security Project. Static Code Analysis. https://www.owasp.org/index.php/Static_Code_Analysis. [Online; accessed 08-October-2014].

[4] Campwood Software. SourceMonitor Version 3.5. http://www.campwoodsw.com/sourcemonitor.html. [Online; accessed 08-October-2014].

[5] GUC Subversion. Shark Cage - Protecting User Interfaces Against Malware. https://svn.hig.no/2014/imt3501/assignment3/Shark%20Cage%20-%20Protecting%20User%20Interfaces%20Against%20Malware.pdf. [Online; accessed 08-October-2014].

[6] Wikipedia. Black-box Testing. http://en.wikipedia.org/wiki/Black-box_testing. [Online; accessed 08-October-2014].