Vulnerabilities in the SecureDesktop application

Mandatory Assignment 4

By: Anna Kaardal - 120918
Per Christian Kofstad - 120916
Synne Gran Østern - 120922

In this assignment we have chosen to mainly focus on static code analysis tools to find vulnerabilities in the SecureDesktop application. We will first explain briefly our method of action. We presume that the reader has gained some background information by going through the article *Shark Cage - Protecting User Interfaces Against Malware*. We will therefore not explain the background of this solution in detail.

Method of action

We started with dividing up 3 different initializing challenges. Learn about tools, learn about Pascal and learn about static code testing. In addition to these the 3 tasks we all did study the article that followed the code, *Shark Cage: Protecting User Interfaces Against Malware.*

In order to get some result from our reading we shared our findings in the group and divided into groups for testing tools and reviewing the code. After that we made a short plan of action for how to deal with the different remarks in the code. More about how we would do our plan of action of the code in the section “Our plan of action for fixing the code”.

Static code analysis

Static code analysis is a method for debugging of programs without executing them(1). Vulnerabilities can be discovered on a early stage, because the code is examined before executing it. This might help reducing the number of code revisions, which will take time and costs, and would be even more expensive on a later stage, because more elements would have to be changed to fix the vulnerabilities.

There are different kinds of static code analysis, and a widely used approach is automated tools. A automated tool aims to uncover and remove vulnerabilities, such as buffer overruns, invalid pointer references and uninitialized variables(2). It will generate warnings or errors when discovering security flaws in the code.

Automated tools have many advantages, they are cost- and time effective, because less people will be needed to manually examine the code. There will be possible to go through fragments of the code rarely controlled, and vulnerabilities hard to find with manual reviews might also be easier detected. It will be able to go through all the code, finding the exact location of a vulnerability, which is very time-saving.

Static code analysis tools do also have some drawbacks. Many false positives will be generated - Vulnerabilities are reported, but they are not actual vulnerabilities. For instance, Findbugs reports to have a false positive rate less than 50 %(3), which should be a good percentage. False negatives - actual vulnerabilities which are not discovered, might also occur, because static code analysis tools will not be able to detect all forms of security flaws, mostly logical flaws. For instance, there are some problems hard to detect without executing the program, like memory leakages(4). Therefore manually follow-up and execution of code will still be important.

Another disadvantage is that different static code analysis tools does not support all programming languages. We spent some time finding tools supported by Pascal. There does not exist as many as for Java, but we found quite a few of different kinds.

In most tools, there are possible to add rules or change options, to further specify which types of findings there should be generated warnings for, and what is going to be ignored. This might help reducing the number of false positives. For instance, in Pascal Analyser you have an options menu where you can choose which files to analyse and which you want to leave out, whether you want variables, parameters, local identifiers and so on to be reported or not.
There are also default options set.

Shark Cage, the article

The article is about the source files we analyse, and describes the concept around the source files and gives a good understanding about the overview concept of the software the source files generate. Although the article uses a lot of words to argue about the concept, it is still relevant in order to understand the purpose of the source files and what things to look for when we do the static code testing.

Our source files generates a program that claims to give a proof of concept that there is possible to create a trusted path for a running application inside the Microsoft Windows operating systems without the need of extra hardware. This tells us that the program should not leak any information in any way, and that this might be a lead in where to start digging in the code analysis.

Pascal / Delphi

Pascal as a programming language were developed and established by Niklaus Wirth in 1970. It was named after Blaise Pascal, a mathematician from the 16th century. The original goal Wirth wanted to succeed with the Pascal language was to:

> "a) make available a language suitable for teaching programming as a systematic discipline based on fundamental concepts clearly and naturally reflected by the language, and b) to define a language whose implementations could be both reliable and efficient on then-available computers."(5)

However, Wirth achieved these goals and more. Pascal became a commercial language and very popular and widely used in the 1970s.(5) Delphi is an extension for Pascal and is often referred to as Object Pascal.(6)

Tools

We found and used several tools in order to create some baselines and data for comparing. After some searching we found that there was both graphical and command based tools to use. Pascal is an old language, it does not seem to

Source Monitor (7)

Source Monitor is an high rated analyse tool that can analyse different languages including Delphi/pascal. It is open source.(7) To begin with it did not tell us much, but after some reading we could see that the information we found contained indicators to where we should start to go through our code.

CodeHealer (8)

CodeHealer is a licensed alternative that you have to pay for to use. We got a trial version and started to see if we could get some useful data out of it.

Pascal Analyser (9)

Pascal analyser is a paid tool, but is available under evaluation licence. It seems to go in detail on many different checks and reports.

Our results from Pascal Analyser

We found that Pascal analyser didn't manage to import more than one module at the time, if we tried to open the modules as a multi module project we constantly got an error. Therefore we needed to look at the number for each of the three modules separately. However when we did this, we found that the results was much the same in all the three projects.

Since we only was able to get an evaluation license of the program we could not see the full list of every report. But we got an indication of what to look for in the code.

None of the modules generated any strong warnings, but all three generated a few warnings. We find a few remarks in the convention compliance and in the code reduction for all three modules but it seems to be much the same type of remarks that shows us throughout the analysis.

The warnings in basically in the category that functions are declared as available as public, but not used outside the unit, or that functions are called as procedures, so the return value is not used. Visibility for functions that do not need to be visible should not be necessary, so to clean up the code we should make this private if it is possible.

The code convention compliance reports shows that there is some class fields that are not declared in the private section. A few cases of empty blocks, identifier with the same string, and unreferenced declared identifiers.

The code reduction report tells us that there are quite a few identifiers that are never used throughout all three modules.

After testing a few other tools, in short, and without having the time to go to much in-depth in the functionality of the tools, we could see that many of the remarks is about the same thing as already discovered. Also we could see that there is a quite high percentage of commentary in some of the files. This was something that we wanted to look more into in the manual code review.

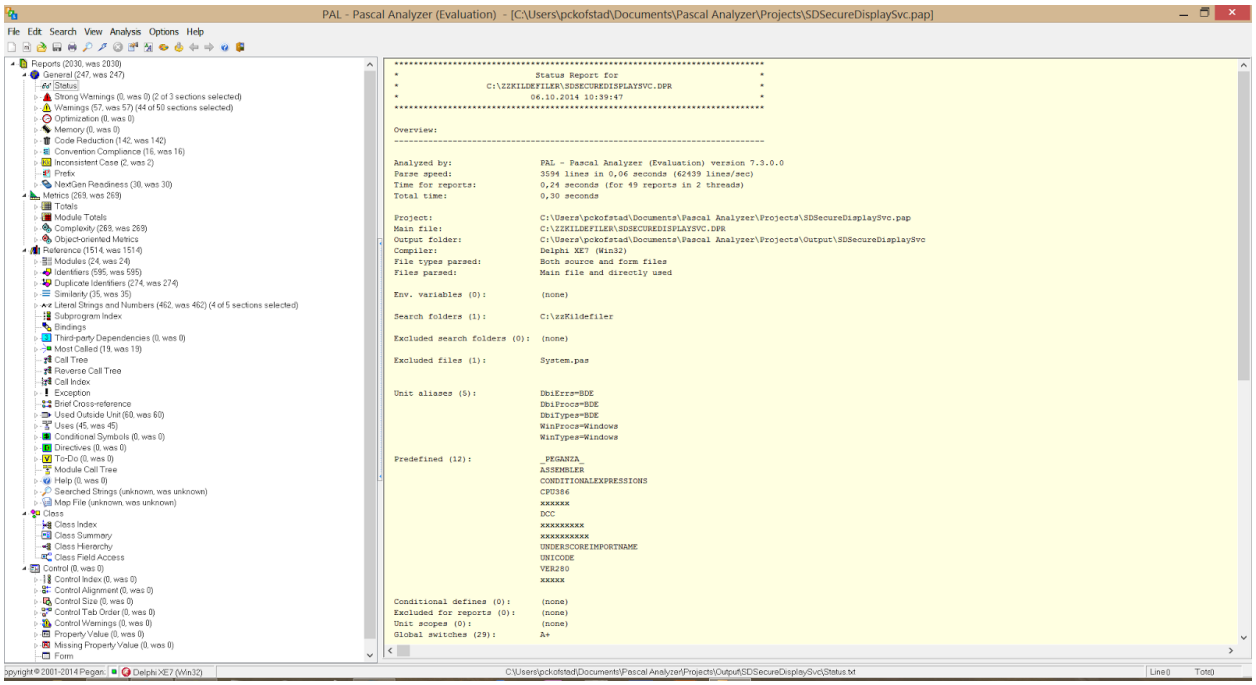

Figure 1: Screenshot from Pascal Analyser

Our results from manual code review

The SecureDesktop application code consist of 17 element, 11 of them are pascal files, 3 Delphi files, 1 readme text, 1 manifest file and 1 xml file. There are in these files minimal or none comments to explain what happens in the code for an outsider who does not know the code. This makes the understanding the code difficult.

The 3 Delphi files

The 3 Delphi files are in the “readme.txt” explained to correspond to the Cage labeller (SDAppInfo), Cage Manager (SDUserSessionManager) and the Cage Service (SDSecureDisplaySvc) from the *Shark Cage - Protecting User Interfaces Against Malware*

article. These three files are also the only programs in the application, the other Pascal files are units. The SDUserSessionManager is the largest file of the 3, with its almost 300 code lines. However many of these code line are commented out. We do not know why this is.

The 11 pascal files
These files are not defined as programs, but as units. In the Pascal language units are subprograms. We will now comment on the files that stands out of the rest.

SDProtocol.pas is not one of the bigger files, but what makes this special is the large amount of global variables. All these are defined as const. Again the reason for why this file was created as it is, is not explained. But we could assume that the developers wanted to have one file with all the global variabels in the application. However this file is not the only one with global variables.

SDSecureDisplaySvcUnit.pas is one of the larger files with its 520 code lines. However, almost half of these code lines are commented away.

SDCommon.pas and *SDInfoSecurity.pas* with their about 400 code line is two of the largest files in the application. There are also little code commented away as well. There are almost 20 global variable declared in the beginning of the *SDCommon.pas* whereas *SDInfoSecurity.pas* have none.

SDInfoProcesses.pas is the largest file with its over 1100 code lines. Very little is commented out.

Our plan of action for fixing the code

The first step in any code review is getting to know the code, and the intention of the different parts of the code. This means studying the code, the comments and if possible, talk to the previous developers. We do this in order to get an impression and a baseline of what the developer was thinking when taking different decisions when designing and developing the code. For example, there could be a reason why the developer chose to have many global variables.

After we have got to know the code we will go through the results from Pascal Analyser, or any other static code analysis tool, looking for strong warnings and warnings, noting these for further examination. We will also examine whether the code is according to Pascal code compliance, and if there is any code that is not used, and does not need to be a part of the source code.

While we go through the remarks from the static code analysis tools, and go briefly through the code manually. The static analysis will presumably generate many false positives, and we have to examine whether they are actual vulnerabilities or not. We will also take a briefly look for possible false negatives, which the tools normally are not able to detect. By this we mean logical programming mistakes, things that an code analyser will not understand is against our intentions.

There are also a lot of code sections out-commented in the code, and we are going to check out if these might be leaved out, or if they have to stay like they are. There might be possible that the code is not yet finished, and the programmer has plans for these unused identifiers and out-commented code.

While we work with fixing the code, or finding out if there is a need to fix the code at all, we will have to make some decisions. Decisions of whether or not there will be vice to modularize more or not, whether we can have fewer global variables or not, whether it is possible to do it faster or easier or not. Some of this questions must be considered according to the future plans for the application. Does the code provide enough security against attacks, is it easy to maintain, if we use time mapping the different parts of the code, should we also use time on updating comments and adding comments where they are not complete enough?

There will be questions that needs discussion, but in order to answer these questions, we will need to have more information on what the code will be used for, by who and how much time and money we have to fix it.

Findings

After the static code analysis, we used a combination of manually code review and searching through the code for specific string matches with the advanced search function in Notepad++ (like array boundaries and external files.) Figure 2 shows for example the result of the array search. We searched for suspicious code and then used project search to search for similar patterns in all the other files.

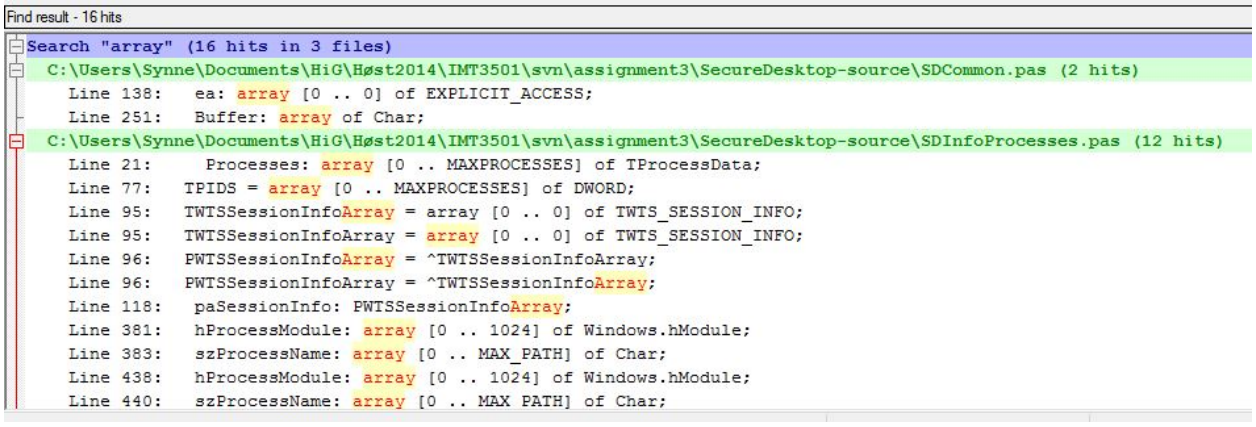

Figure 2: Screenshot of the Notepad++ search on “array”

Hardcoded paths

Some of the file names are hardcoded, and is not relative to the program-location file. This might cause error in the program, or place files on locations where they are not supposed to be stored, in cases where the program is stored on another computer.

File	Ln	Command
SDCommon.pas	68	SDLogFileName = 'C:\Projects\WinStaTest\SDLogFile-';
SDCommon.pas	111	StrPCopy(pszAppDataPath, 'C:\Users\hannol\AppData\Roaming');
SDInfoSecurity.pas	333	if SDCreateProcessWithTokenOnDesktop('C:\Windows\system32\calc.exe ', '',
DSeureDisplaySvcUnit.pas	368	SDCreateProcessWithTokenOnDesktop('C:\Windows\system32\charma p.exe', hUsualUserToken, SecureViewerDesktopName);
SDSeureDisplaySvcUnit.pa s	491	SDCreateProcessWithTokenOnDesktop('C:\Windows\system32\notepad .exe', hSecureViewerGroupEnabledToken, SecureViewerDesktopName);

We also found 8 other examples, but they was commented away, so we see them as irrelevant at this point.

Hardcoded duplicated variables

Some variables that clearly are declared and used many places are hardcoded. It will be safer using variables instead, ensuring that every instance is changed, for instance if the developer want to change the code.

File	Command
SDInfoProsesses.pas	Line 10: MAXPROCESSES = 2048; Line 152: cbName := 2048 + 1; Line 242: cbName := 2048 + 1; Line 292: cbName := 2048 + 1; Line 154: cbReferencedDomainName := 2048 + 1; Line 244: cbReferencedDomainName := 2048 + 1; Line 294: cbReferencedDomainName := 2048 + 1;
SDInfoSecurity.pas	Line 409: cbName := 2048 + 1; Line 411: cbReferencedDomainName := 2048 + 1;
SDInfoProsesses.pas	Line 565: nPrivilegeNameSize := 255; Line 571: nPrivilegeNameSize := 255; Line 752: nPrivilegeNameSize := 255; Line 756: nPrivilegeNameSize := 255; Line 839: nPrivilegeNameSize := 255; Line 845: nPrivilegeNameSize := 255; Line 567: nPrivilegeDisplayNameSize := 255; Line 572: nPrivilegeDisplayNameSize := 255; Line 841: nPrivilegeDisplayNameSize := 255; Line 846: nPrivilegeDisplayNameSize := 255;

There are a possible buffer overflow in SDCommon.pas.

- On line 270 in SDCommon.pas Buffer is converted to string and copied into AMessage, without being allocated, or converted.
- Buffer is declared as array of char on line 251.
- The AMessage is copied into itself with allocation of memory as long as it already is on line 273.

There are a possible buffer overflow in SDInfoProcess.pas.

- Line 21 the array *Process* gets declared to the size of MAXPROCESS, which is 2048.
- This array is then used in the *SDEnumerateProcesses* in line 374. But the code does not check the size of this array to prevent buffer overflow.

There is a possible vulnerability in SDCommon.pas

- In line 23 a bmp-file is added to the variable *BackgroundBitmapFileName*. We do not know what kind of picture this is, but there might be a possibility to have a malicious bmp-file.

To sum the manual search up. We did find some possible weaknesses in the code. However we are not sure if it is possible to exploit all our findings to achieve some sort of security breach.

Summary of our work

In this assignment we chosed to mainly focus on static code analysis tools to find vulnerabilities in the SecureDesktop application, but we did also a manually review. We presumed that the reader had gained some background information by going through the article: *Shark Cage - Protecting User Interfaces Against Malware*. We needed to check out what static code analysis is, then we needed to find tools, test the tools, and discuss and draw conclusions of our findings.

Conclusion

We found that there are quite a lot of static code analysis tools for Delphi/Pascal, but many of them is licensed so you need to pay to use them fully. After using the different tools, we tested and found that the results from the analysis is good indication on specific errors and programming trends throughout the code. However, the analysis cannot see logical programming mistakes that the developer has done. This means that whether or not you use an static analysis tool, you will still have to go through your code manually afterwards in order to check it for errors. We found it also very challenging to analyse and understand code that had so little commenting and explanation about the reason and idea behind this solution. After going through the code in our manual review, we found some vulnerabilities that we would have corrected. These are mainly concerned hardcoded values and possible buffer overflows.

References

1. Margaret Rouse, Static analysis (static code analysis), http://searchwindevelopment.techtarget.com/definition/static-analysis, Timestamp: 02.10.14
2. Michal Cobb, Static source code analysis tools: Pros and cons, http://searchsecurity.techtarget.com/answer/Static-source-code-analysis-tools-Pros-and-cons, Timestamp: 07.10.14
3. FindBugs Fact Sheet, http://findbugs.sourceforge.net/factSheet.html, Timestamp: 04.04.14
4. Karpov Andrey, Static Code Analysis, http://www.codeproject.com/Tips/344663/Static-code-analysis, Timestamp: 07.10.14
5. Bill Catambay. *The Pascal Programming Language* http://pascal-central.com/ppl/index.html 05.09.2001
6. Ken Doyle, Apple Computer. *Introduction to Object Pascal* http://www.mactech.com/articles/mactech/Vol.02/02.12/ObjectPascal/index.html Timestamp: 07.10.2014
7. Source Monitor (http://www.campwoodsw.com/)
8. CodeHealer (http://www.socksoftware.com/)
9. Pascal Analyser (http://www.peganza.com/products_pal.htm)