Static Analysis of C source code

Laboratory for the class "Security Verification and Testing" (01TYASM/01TYAOV)
Politecnico di Torino – AY 2023/24
Prof. Riccardo Sisto

prepared by:
Riccardo Sisto (riccardo.sisto@polito.it)

v. 1.0 (21/11/2023)

Contents

1 Purpose of this laboratory

The purpose of this lab is to make experience with static source code analysis tools for the C/C++ languages. More specifically, two tools will be experimented: a simple lexical scanner (flawfinder) and a more sophisticated commercial static analysis tool (PVS-Studio). As the two tools have not only different features but also different coverage of vulnerabilities, their combined use is recommended. For the installation of the tools, please refer to the 00Lab_GettingStarted_SVT_v104.pdf guide.

All the material necessary for this lab can be found in the course web pages on didattica.polito.it, Materiale Didattico, 04Lab_SC folder.

2 Getting started with flawfinder and PVS-Studio

Before starting with the real exercises, let us make some tests to check the tools are properly set.

Running flawfinder to reproduce some of the results shown in the classroom

Run flawfinder on the CWE121.c file taken from the NIST Juliet Test Suite for C/C++, and check that you get the expected 3 hits that we saw in the classroom (you should get 3 hits if you use version 2.X, 4 hits if you use older versions).

Getting Started with PVS-Studio

Make sure you have run the following command to install the free academic license:

```
pvs-studio-analyzer credentials PVS-Studio Free FREE-FREE-FREE-FREE
```

Note that, when working on the Labinf machines, the PVS-Studio free license should remain stored in your home directory after the execution of the command, so you will not have to run it again in future sessions.

As we use PVS-Studio from the command line, some bash scripts are provided to simplify running PVS-Studio. They are included in the zip file named pvs-script.zip. Extract this archive in your home directory. The scripts will be copied to your bin directory, which will be created if not present yet. In order to complete the setup add the bin directory to the PATH if not yet included. This can be done by adding the following line to the .bashrc file in your home directory:

```
export PATH=$PATH:[home directory]/bin
```

where [home directory] is your home directory.

The pvs-addcomment script can be used to add the necessary comment to all the .c files in the current directory. The pvs-run script can be used to run PVS-Studio. You must run it with the same command-line arguments that you use for the 'make' command when you compile the program. The report is generated in HTML format (in the htmlreport directory). If you want to change the options used to run PVS-Studio you can edit the pvs-run script. The pvs-clean script makes a cleaning by removing the files generated by PVS-Studio, including the result files. It is called automatically by pvs-run before running PVS-Studio.

Running PVS-Studio to reproduce some of the results shown in the classroom

Run PVS-Studio on the CWE121.c file taken from the NIST Juliet Test Suite for C/C++, by entering the following commands from the CWE121 directory (note that the makefile in this case requires no arguments):

```
pvs-addcomment
pvs-run
```

Check that the analysis proceeds without errors and that you get the html report containing a single entry, as shown in the classroom.

Now, try to run PVS-Studio from the demonstration web site:

```
https://pvs-studio.com/en/pvs-studio/godbolt/
```

Here, you can edit the C code that is in the left hand side text area. When you change the code, the tool runs automatically and you can see the new results on the right hand side. If you prefer, you can open an alternative view, by clicking on 'Edit on Compiler Explorer'. Try to fix the sample code that is displayed and check that the errors reported disappear. The archive of the lab contains a very simple test file that contains a classical format string vulnerability. It is in the test1 directory. Copy the contents of the file and paste the code into the left-hand size window, by overwriting the previous code. The format string vulnerability should be pointed out by PVS-Studio. Fix the code and check that PVS-Studio does not report the error after the fix.

3 Static Analysis with Flawfinder and PVS-Studio

3.1 Analyzing other simple examples

Use Flawfinder and PVS-Studio to analyze the other simple examples found in the lab material (test2 and test3). For each one of them, run Flawfinder and PVS-Studio. Then, analyze each reported issue and decide if it is a true positive (TP) or a false positive (FP). Write a report of your findings containing, for each reported issue, the classification as TP or FP and an explanation of each decision. Finally, sort the TP according to their severity.

Solution

test2.c
flawfinder, line 46: FP (the destination local cannot overflow because it initially holds a correctly terminated 10 bytes long string, while the source buf initially holds a correctly terminated string which is at most 127 bytes long; hence, the string stored in local after the concatenation is at most 127+10=137 bytes long, which can be stored , including its string terminator, in the allocated 138 bytes)
flawfinder, line 47: TP (system executes a command that is not trusted, because part of it comes from buf, which contains data read from a socket)
flawfinder, line 17: TP (buf can overflow at line 50, see line 50)
flawfinder, line 42: FP (the buffer cannot overflow, see the FP at line 46)
flawfinder, line 43: FP (the buffer is written only at line 48 and it cannot overflow because, although the maximum number of bytes to be copied is set to 140, which is larger than the destination size, no more than 138 bytes will be copied, because at line 48 the source local holds a correctly terminated string which is at most 138 bytes long including the terminator)
flawfinder, line 45: FP (see line 42)
flawfinder, line 26: FP (read is used correctly)
flawfinder, line 48 and PVS-Studio 50: FP (see line 43)
flawfinder, line 50 and PVS-Studio 52: TP (at line 50 the source log can hold a correctly terminated string of length up to 138 bytes, including the terminator, while the destination is only 128 bytes long; as the contents of log partially come from buf, which contains data read from a socket, the severity of this vulnerability is high)
flawfinder, line 51: TP (buf is 0-terminated, but it can overflow, see line 50; hence strlen(buf) may read outside the buf bounds and cause a crash)

test3.c
flawfinder, line 37: FP (inputbuffer cannot overflow because it is written only at lines 63 and 69 and in both cases data are written within the bounds)
flawfinder, line 71: TP (The size of inputbuffer is 3 * sizeof(data) + 2. If we assume int is 4 bytes long, the size is 14. With this size, it is possible to represent integers up to 999999999999, which may cause an integer overflow when atoi is executed. Since at line 95 data is used as index in a write operation on buffer, the consequence is that a byte with value 1 can be written outside the buffer bounds. This could be used by an attacker to write value 1 into arbitrary memory locations, possibly causing a crash or unintended behavior)

TP, sorted in decreasing severity order:
high: test2.c 47,17(50): the attacker can execute arbitrary code on the target.
medium: test3.c 71 (and 95): the attacker could cause a crash or unintended behavior.
low: test2.c 51: the attacker could cause a crash.

3.2 Analysis and Fix of a real vulnerable code

An implementation of the UNIX file() command was affected by a buffer overflow vulnerability reported in a CVE. This exercise consists of analyzing the vulnerable code to find this vulnerability. In the material for the lab, you can find the package with the sources of the version of the software affected by the vulnerability. Run flawfinder on the file readelf.c, which is the file containing the vulnerability. Analyse the hits returned by flawfinder and classify them into true positives (TP) and false positives (FP). For each one of them, explain the reason for your classification.

Solution

lines 81,100,121,333: FP (the statically sized array is used consistently in the scope of the definition)
line 535: TP (the number of bytes copied by memcpy, i.e. variable descsz, is not properly checked

because the condition (descsz >= 4 || descsz <= 20) is always true. As the contents of the source as well as of descsz come from a read operation on the file, an attacker could control what is written outside the buffer boundary.
lines 559,626,656: FP (the destination can always hold the source data because the number of bytes copied is the size of the destination)
line 720: FP (the statically sized array is used consistently in the scope of the definition)
line 723: TP (this is a potential buffer overflow because the number of bytes copied by memcpy, i.e. variable descsz, is not checked in the function; depending on how the function is called, this could be a vulnerability or not)
line 954: FP (the destination can always hold the source data)
line 996: FP (the statically sized array is used consistently in the scope of the definition)
line 1040: FP (the destination can always hold the source data)
lines 1214,1327: FP (the statically sized array is used consistently in the scope of the definition)
lines 1340,1352,1366: FP (the destination can always hold the source data)
lines 1477,1478: FP (the statically sized array is used consistently in the scope of the definition)
line 1578: FP (the statically sized array is not used)
line 1331: FP (read cannot write outside the buffer)
line 1350: TP (as in the code there is no evidence that the string pointed to by p is null-terminated, this could be a vulnerability, but further analysis is necessary)

In summary, we have found 3 potential vulnerabilities. The one reported in the CVE (which is CVE-2017-1000249) is the vulnerability at line 535

Now, try to use PVS-Studio for the analysis of the code. Before being able to compile the code by the 'make' command it is necessary to generate the makefile, by running the following commands (see README.DEVELOPER):

```
autoreconf -f -i
./configure --disable-silent-rules
```

Then, you can check that the code can be compiled by running

```
make -j4
```

another preliminary operation before running PVS-Studio is to insert the two special comment lines at the beginning of each C file. This can be done by means of the pvs-addcomment script, after having moved into the src directory:

```
cd src
pvs-addcomment
```

Finally, PVS-Studio can be run by running

```
make clean
pvs-run -j4
```

Note that whenever you want to repeat the analysis you need to clean the project, because PVS-Studio can only analyze the files that are actually compiled (make will automatically avoid the compilation of files if the result of compilation is up to date). Look at the issues reported by PVS-Studio about the readelf.c file. What can we say about PVS-Studio's ability to find the vulnerability in this file?

Solution

PVS-Studio did not report the vulnerability at line 535 (so, technically, it is a false negative for PVS-Studio). However, PVS-Studio reports the cause of the vulnerability, i.e. the error in the boolean expression of the condition that controls the vulnerable memset operation. So, by fixing this error, the vulnerability is also fixed.

Find a fix for the vulnerability and write a patched version of the file. Then use the tools to analyze the code again.

Solution

The vulnerability can be fixed by correcting the wrong condition into (descsz >= 4 && descsz <= 20), which guarantees that descsz is less than 20, i.e. that less than 20 bytes are copied) (see readelf_fixed.c in the solution folder)