Detecting Open Source Project unicorn with FlawFinder

Shandian SHEN
u201911729@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, Hubei, CHINA

ABSTRACT

In this study, I conducted a static analysis on the open source software, Unicorn, using FlawFinder, a static vulnerability detection tool for C/C++. The project, which has more than 10 stable versions and 6k stars on GitHub, is known for its multi-architecture, lightweight design, and support for Windows and *nix systems. The analysis covered 12 historical versions of the project from v1.0.3 to v2.0.1.post1. After that, I used a python script to parse the analysis results of FlawFinder and separate the types and quantities of CWEs. I also used the git diff command to compare the analysis files of different versions to determine if the vulnerabilities between versions had been fixed.

Through the use of FlawFinder, I discovered that the number of issues and vulnerabilities detected by the static analysis tool increased over time. Upon manual review, I found five different findings of varying depth and types. However, I also noticed that most of the vulnerabilities detected by the static analysis tool were not addressed by the developers in the subsequent versions. Further review revealed that the tool had a high false positive rate, which was disappointing and frustrating.

KEYWORDS

static analysis, FlawFinder, C++

ACM Reference Format:
Shandian SHEN. 2022. Detecting Open Source Project unicorn with FlawFinder. In *Proceedings of Program Analysis and Security (PAS'22).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

This study used the FlawFinder static vulnerability detection framework for C++ to analyze the open source software unicorn, a lightweight, multi-platform, multi-architecture CPU emulator framework based on QEMU. The project, which has more than 10 stable versions and 6k stars on GitHub, is known for its multi-architecture, lightweight design, implementation in pure C language, native support for Windows and Unix-like systems, high performance through Just-In-Time compilation, support for fine-grained instrumentation at various levels, and thread-safety. The analysis covered 12 historical versions of the project from v1.0.3 to v2.0.1.post1, and found that while there were no fatal vulnerabilities, there were still design defects and deficiencies present. After actual testing and verification, it was discovered that many of these design defects had not been repaired and improved in the versions from v1.0.3 to v2.0.1.post1, leaving the possibility of exploitation by attackers. FlawFinder was used to generate an HTML vulnerability report for each version of the project for easy viewing.

All analysis results can be found online[2].

PAS'22, Nov 11-Dec 31, 2022, Wuhan, China
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 BACKGROUND

Automatic vulnerability mining based on pattern recognition can be divided into two categories: source code-based scanning analysis and disassembly code-based scanning analysis. Source code-based scanning analysis utilizes common syntax and lexical analysis rules, as well as control flow and data flow analysis, to track the data and logic of the source code. Disassembly code-based scanning analysis, on the other hand, uses static taint analysis, symbol execution, directed graph analysis, and other methods or specific detection rules to classify and analyze vulnerabilities in the source code. Automatic vulnerability mining often relies on symbolic execution to build an automatic vulnerability detection mechanism based on specific operations or rules, such as intermediate results or logical reasoning. This approach is effective for small batches of vulnerabilities with similar types and structures, but may not be suitable for detecting multiple types of vulnerabilities in the source code. While automatic vulnerability mining can effectively detect vulnerabilities in a clear rule mode, it also has limitations:

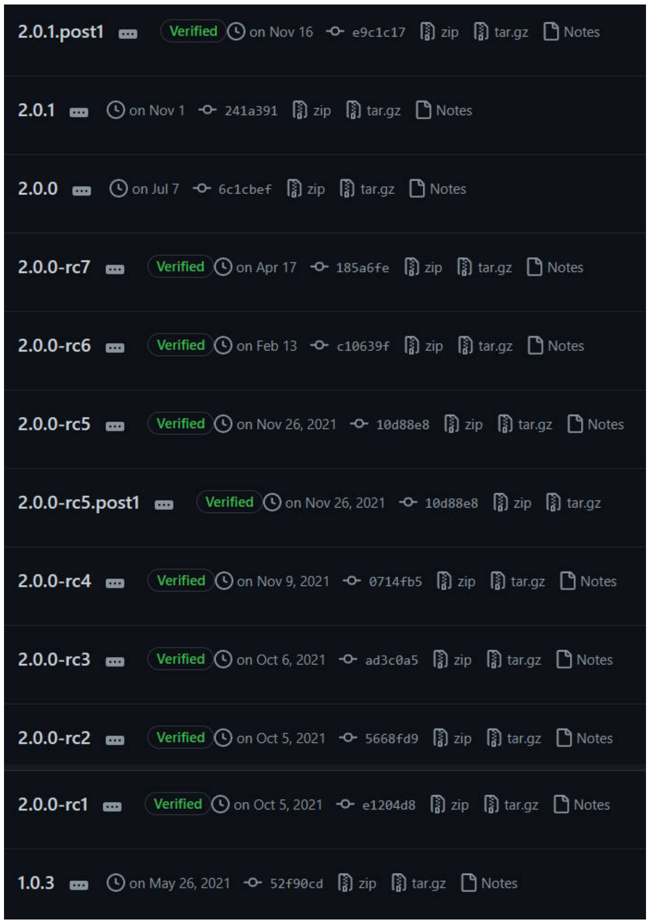
- Automatic vulnerability mining may have a high false positive and false negative rate for the same type of vulnerabilities in different modes, making it less adaptable.
- As vulnerability types and forms constantly evolve, traditional rules may not be sufficient for detecting new vulnerabilities.

FlawFinder is an open-source static scanning and analysis tool for C/C++ that uses a python-based automatic vulnerability detection system. It searches for simple defects and vulnerabilities in C/C++ code by matching them against an internal dictionary database. Unlike other tools, FlawFinder does not require the code to be compiled, making it easy and fast to use. It is also free and compatible with the Common Weakness Enumeration (CWE) system, earning the CII Best Practices "passing" badge. FlawFinder can generate output in various formats, including TXT, HTML, CSV, and JSON (Sarif format). However, the author of FlawFinder also acknowledges that no tool can detect all types of vulnerabilities and that FlawFinder has its own limitations.

3 EXPERIMENT SETUP

3.1 Experiment Data

I collected 12 historical versions of unicorn from v1.0.3 to v2.0.1.post1, whose version numbers are shown in Figure 1.



```
2.0.1.post1      Verified  on Nov 16           e9c1c17  zip  tar.gz  Notes
2.0.1                      on Nov 1            241a391  zip  tar.gz  Notes
2.0.0                      on Jul 7            6c1cbef  zip  tar.gz  Notes
2.0.0-rc7        Verified  on Apr 17           185a6fe  zip  tar.gz  Notes
2.0.0-rc6        Verified  on Feb 13           c10639f  zip  tar.gz  Notes
2.0.0-rc5        Verified  on Nov 26, 2021     10d88e8  zip  tar.gz  Notes
2.0.0-rc5.post1  Verified  on Nov 26, 2021     10d88e8  zip  tar.gz
2.0.0-rc4        Verified  on Nov 9, 2021      0714fb5  zip  tar.gz  Notes
2.0.0-rc3        Verified  on Oct 6, 2021      ad3c0a5  zip  tar.gz  Notes
2.0.0-rc2        Verified  on Oct 5, 2021      5668fd9  zip  tar.gz  Notes
2.0.0-rc1        Verified  on Oct 5, 2021      e1204d8  zip  tar.gz  Notes
1.0.3                      on May 26, 2021     52f90cd  zip  tar.gz  Notes
```

Figure 1: 12 history versions of unicorn software.

3.2 The Detail Information of unicorn software

As shown above, unicorn has 12 historical versions from v1.0.3 to v2.0.1.post1. During the code review, I discovered that v2.0.0-rc5.post1 is identical to v2.0.0-rc5, so the actual number of versions analyzed is 11.

The number of files and lines of code in each version of unicorn that I collected, which was obtained using the Gitstats tool, is shown in Table 1.

I used git to switch between different versions. For example, the following command can be used to switch to v2.0.1.post1:

- git reset --hard e9c1c17

Because I was in an Ubuntu environment, I used the following command to install Gitstats:

- sudo apt install gitstats

And I extracted the number of files and lines of code with it. For example, we can use the commands:

- gitstats ../unicorn/ ../html/unicorn-1.0.3/

3.3 The Reason For Selecting This Project

As mentioned previously, unicorn software is a lightweight, multi-platform, multi-architecture CPU emulator framework. It has features such as being implemented in pure C language, native support for Windows and Unix-like systems, high performance through Just-In-Time compilation, support for fine-grained instrumentation at various levels, and thread-safety. These characteristics make unicorn an ideal candidate for a static analysis study.

The main reason for conducting this static analysis of unicorn is that I am using the open source tool for a separate project. By understanding the past and current features of unicorn, I can better utilize it in my project.

3.4 Experiment Process

3.4.1 *Installing and Checking the Version of FlawFinder.*

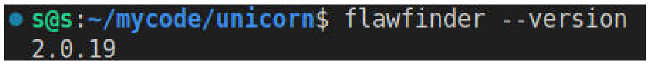
FlawFinder can be installed in several ways, such as downloading it from the official website or using an existing Python environment on the computer. Since my computer has Python 3.9 installed, I can use the python command directly:

- pip install flawfinder

After installation, we can also use the command:

- flawfinder --version

to view the relevant versions of FlawFinder, as shown in Figure 2:



```
s@s:~/mycode/unicorn$ flawfinder --version
2.0.19
```

Figure 2: Check the version of FlawFinder

3.4.2 *Scanning unicorn software with FlawFinder.*

FlawFinder has various scanning methods. For example, we can use the commands:

- flawfinder --csv > test-result.csv test.c

to scan a single test.c file and store the scanning result in the test-result.csv format. The result will be saved in the same directory as the test.c file. We can also specify different output formats, such as:

- flawfinder --html > test-result.html test.c

to generate HTML format files. We can also use the following command:

- flawfinder --html --context test.c > test-results.html

to print the code content.

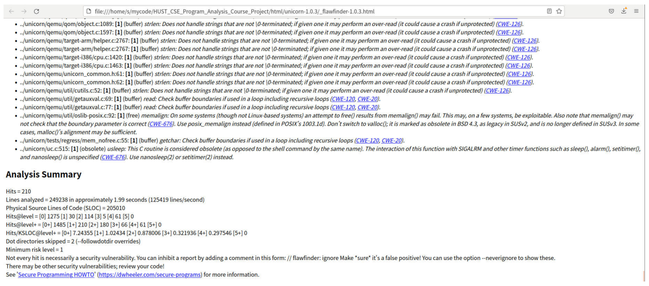
Since scanning a single file is inefficient, this study uses another scanning method of FlawFinder, which is global scanning. For example, we can use the command as followed:

- flawfinder --html > ../html/flawfinder-1.0.3.html ../unicorn

The FlawFinder will scan all the files under the unicorn folder and save the output results in the html format. We can open the HTML file to view the example result shown in Figure 3:

Table 1: Number of files and lines contained in the historical version of unicorn

Version of unicorn	Number of files contained in unicorn	Number of lines contained in unicorn
V1.0.3	859	318642
V2.0.0-rc1	879	490183
V2.0.0-rc2	879	490191
V2.0.0-rc3	877	490224
V2.0.0-rc4	865	495744
V2.0.0-rc5	865	496644
V2.0.0-rc6	923	524608
V2.0.0-rc7	927	526412
V2.0.0	955	545594
V2.0.1	952	546266
V2.0.1.post1	952	546275


file:///home/s/mycode/HUST_CSE_Program_Analysis_Course_Project/html/unicorn-1.0.3/_flawfinder-1.0.3.html

```
• ../unicorn/qemu/qom/object.c:1089: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/qom/object.c:1597: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/target-arm/helper.c:2767: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/target-arm/helper.c:2767: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/target-i386/cpu.c:1420: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/target-i386/cpu.c:1463: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/unicorn_common.h:61: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/unicorn_common.h:62: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/util/cutils.c:52: [1] (buffer) strlen: Does not handle strings that are not \0-terminated; if given one it may perform an over-read (it could cause a crash if unprotected) (CWE-126).
• ../unicorn/qemu/util/getauxval.c:69: [1] (buffer) read: Check buffer boundaries if used in a loop including recursive loops (CWE-120, CWE-20).
• ../unicorn/qemu/util/getauxval.c:77: [1] (buffer) read: Check buffer boundaries if used in a loop including recursive loops (CWE-120, CWE-20).
• ../unicorn/qemu/util/oslib-posix.c:92: [1] (free) memalign: On some systems (though not Linux-based systems) an attempt to free() results from memalign() may fail. This may, on a few systems, be exploitable. Also note that memalign() may
  not check that the boundary parameter is correct (CWE-676). Use posix_memalign instead (defined in POSIX's 1003.1d). Don't switch to valloc(); it is marked as obsolete in BSD 4.3, as legacy in SUSv2, and is no longer defined in SUSv3. In some
  cases, malloc()'s alignment may be sufficient.
• ../unicorn/tests/regress/mem_nofree.c:55: [1] (buffer) getchar: Check buffer boundaries if used in a loop including recursive loops (CWE-120, CWE-20).
• ../unicorn/uc.c:515: [1] (obsolete) usleep: This C routine is considered obsolete (as opposed to the shell command by the same name). The interaction of this function with SIGALRM and other timer functions such as sleep(), alarm(), setitimer(),
  and nanosleep() is unspecified (CWE-676). Use nanosleep(2) or setitimer(2) instead.

Analysis Summary

Hits = 210
Lines analyzed = 249238 in approximately 1.99 seconds (125419 lines/second)
Physical Source Lines of Code (SLOC) = 205010
Hits@level = [0] 1275 [1] 30 [2] 114 [3] 5 [4] 61 [5] 0
Hits@level+ = [0+] 1485 [1+] 210 [2+] 180 [3+] 66 [4+] 61 [5+] 0
Hits/KSLOC@level+ = [0+] 7.24355 [1+] 1.02434 [2+] 0.878006 [3+] 0.321936 [4+] 0.297546 [5+] 0
Dot directories skipped = 2 (--followdotdir overrides)
Minimum risk level = 1
Not every hit is necessarily a security vulnerability. You can inhibit a report by adding a comment in this form: // flawfinder: ignore Make *sure* it's a false positive! You can use the option --neverignore to show these.
There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO' (https://dwheeler.com/secure-programs) for more information.
```

Figure 3: An example scanning result of Flawfinder.

To simplify these processes, I wrote two shell scripts, called `switch.sh` and `build.sh`. Simply running `build.sh` will extract the results under the `html` folder.

3.4.3 Data Processing. I wrote some simple python scripts to help parse the analysis results of FlawFinder and separate the types and quantities of CWEs.

Then compare the analysis files of different versions with the `git diff` command to manually determine whether the vulnerabilities between versions have been repaired.

4 EXPERIMENT RESULTS

4.1 Increasing Issues

In the experimental results section, I first counted the number of issues detected by FlawFinder for each version of unicorn, as shown in Figure 4.

As shown in the above figure, there is a trend of increasing issues in the newer versions of unicorn. Specifically, there is a significant increase in issues between the two versions v1.0.3 and v2.0.0-rc1, and between v2.0.0-rc5 and v2.0.0-rc6.

4.2 Increasing Vulnerabilities

FlawFinder categorizes the detected vulnerabilities according to CWE classification standards, so I counted the number of different vulnerabilities present in each version and the results are shown in Table 2.

Some issues may cause multiple types of CWE vulnerabilities, so the number of CWE statistics is greater than the number of detected issues.

The results of the FlawFinder analysis show that the number of vulnerabilities detected in each version of unicorn has an increasing trend. This trend is consistent with the increasing number of issues found in the updated versions of the software. It is clear that the CWE vulnerabilities in the newer versions of unicorn have also increased, indicating that there is still room for improvement in the software's security. Further analysis and testing is necessary to identify and address these vulnerabilities to ensure the software is secure and reliable for users.

As can be seen, the most common vulnerabilities are CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) and CWE-120 (Buffer Copy without Checking Size of Input). These two vulnerabilities account for more than half of the total vulnerabilities detected in all versions of unicorn. Other common vulnerabilities include CWE-134 (Use of Externally-Controlled Format String), CWE-362 (Concurrent Execution using Shared Resource with Improper Synchronization) and CWE-367(Time-of-check Time-of-use Race Condition).

4.3 Empirical Findings

I compared the update instructions for each version release to determine if the FlawFinder detection results were noticed and addressed by the developers.

It should be noted that unicorn is based on qemu, so a significant portion of the issues and vulnerabilities detected are not from unicorn itself, but from the qemu source code.

4.3.1 Static analysis tools may not receive adequate attention. In my analysis of Unicorn software, I discovered that a significant number of the issues identified by FlawFinder were not addressed by the developers during the version iteration. This suggests that static analysis tools may not receive sufficient consideration in the development process.

4.3.2 Correlation with Code Complexity and Modifications. I observed a positive correlation between the number of bugs detected by FlawFinder and the complexity of the code as well as the number

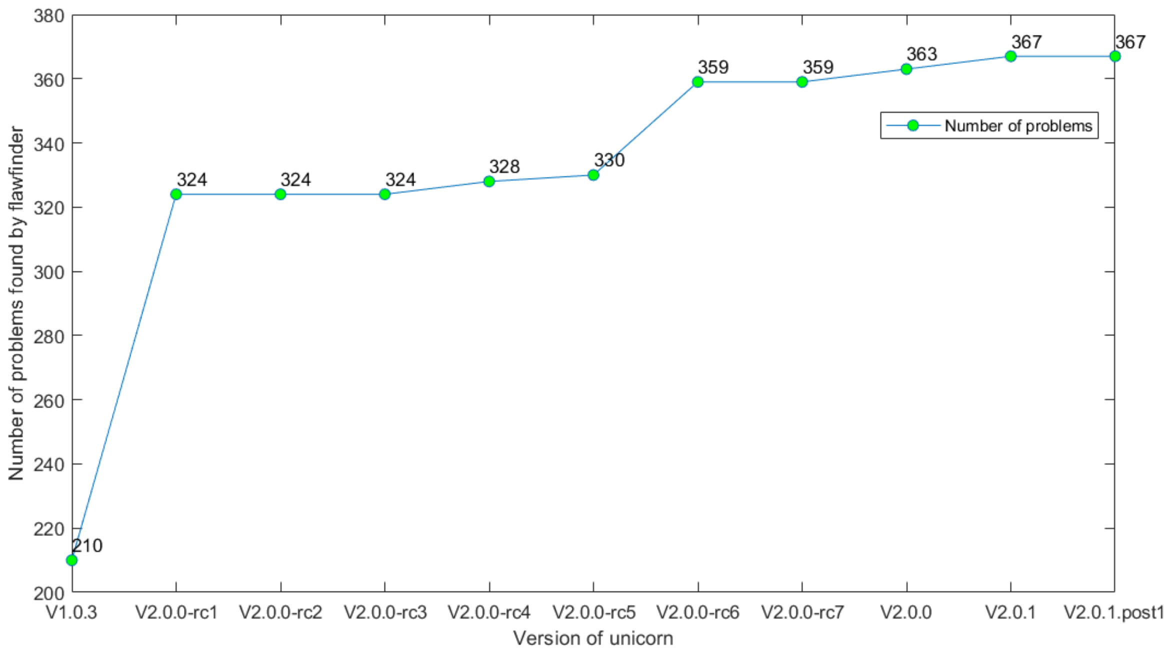


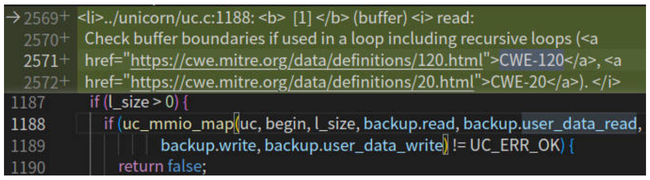
Figure 4: Number of problems found by FlawFinder in the historical version of unicorn.

Table 2: Number of CWE contained in the historical version of unicorn

Version of unicorn All	CWE-20	CWE-78	CWE-119	CWE-120	CWE-126	CWE-134	CWE-190	CWE-327	CWE-362	CWE-367	CWE-676	CWE-807
V1.0.3 285	10	2	48	109	16	39	1	4	35	18	3	0
V2.0.0-rc1 432	16	3	62	208	18	30	3	5	51	31	3	2
V2.0.0-rc2 432	16	3	62	208	18	30	3	5	51	31	3	2
V2.0.0-rc3 432	16	3	62	208	18	30	3	5	51	31	3	2
V2.0.0-rc4 440	16	3	66	212	18	30	3	5	51	31	3	2
V2.0.0-rc5 444	18	3	66	213	18	31	3	5	51	31	3	2
V2.0.0-rc6 490	18	3	69	222	22	32	3	5	66	45	3	2
V2.0.0-rc7 490	18	3	69	222	22	32	3	5	66	45	3	2
V2.0.0 495	19	3	69	224	24	32	3	5	66	45	3	2
V2.0.1 499	19	3	69	228	24	32	3	5	66	45	3	2
V2.0.1.post1 499	19	3	69	228	24	32	3	5	66	45	3	2

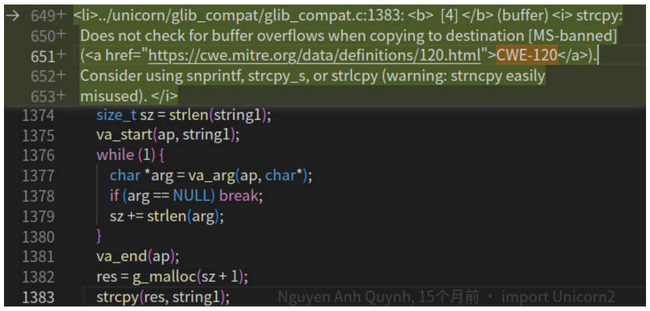
of code modifications. This indicates that more complex and frequently modified code is more likely to have vulnerabilities. It can also be used as a reference for developers to determine the impact of code changes on the stability and security of the software.

4.3.3 Not all vulnerabilities still exist. Although FlawFinder detected an increase in vulnerabilities over time, it does not necessarily mean that all previous vulnerabilities had not been fixed. Upon manual review, I found that some errors detected in earlier versions had been addressed and resolved by developers.



```
<li>../unicorn/uc.c:1188: <b>  [1] </b> (buffer) <i> read:
 Check buffer boundaries if used in a loop including recursive loops (<a
 href="https://cwe.mitre.org/data/definitions/120.html">CWE-120</a>, <a
 href="https://cwe.mitre.org/data/definitions/20.html">CWE-20</a>). </i>
  if (l_size > 0) {
      if (uc_mmio_map(uc, begin, l_size, backup.read, backup.user_data_read,
              backup.write, backup.user_data_write) != UC_ERR_OK) {
          return false;
```

Figure 5: One of the False Positives found in uc.c.



```
<li>../unicorn/glib_compat/glib_compat.c:1383: <b>  [4] </b> (buffer) <i> strcpy:
 Does not check for buffer overflows when copying to destination [MS-banned]
 (<a href="https://cwe.mitre.org/data/definitions/120.html">CWE-120</a>).
 Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily
 misused). </i>
    size_t sz = strlen(string1);
    va_start(ap, string1);
    while (1) {
        char *arg = va_arg(ap, char*);
        if (arg == NULL) break;
        sz += strlen(arg);
    }
    va_end(ap);
    res = g_malloc(sz + 1);
    strcpy(res, string1);        Nguyen Anh Quynh, 15个月前 • import Unicorn2
```

Figure 6: One of the False Positives found in glib_compact.c.

For example, the `vprintf` function vulnerability still exists in five instances in v1.0.3, but is only present in one instance in v2.0.1.post1.

4.3.4 *Not take into account the correction of obsolete functions.*

Developers often do not consider updating functions as long as there are no obvious security risks associated with using obsolete functions. For example, the `usleep` and `memallign` functions are retained from v1.0.3 to v2.0.1.post1.

4.3.5 *Lots of false positives in the tool.*

CWE-120 is the most detected, so I selected it for manual review. This CWE is related to the buffer check. Upon manual review of the CWE-120 vulnerabilities detected in the `uc.c` and `glibcompact/glib_compact.c` files, I found that six of the detected CWE-120 vulnerabilities were false positives. Among them, the three CWE-120 vulnerabilities in the `uc.c` file were very obvious false positives, while the three CWE-120 vulnerabilities in the `glib_compact.c` file were also false positives but indeed worth further investigation.

The six sampling results are all false positives, which frustrates me and challenges the tool's ability to analyze the code context.

Figure 5 shows a false positive found in `uc.c`, and Figure 6 shows a false positive found in `glib_compact.c`.

5 CONCLUSION

In this study, I used FlawFinder to perform static analysis on the open source software project, Unicorn, and presented the detailed results of the analysis. Through this process, I gained a thorough understanding of how to use FlawFinder, a static vulnerability detection tool. The results showed that even for popular open source projects, there are still many vulnerabilities and design flaws present. FlawFinder also has its own limitations as a detection tool. As shown in the study "How Many of All Bugs Do We Find? A Study of Static Bug Detectors"[1] it is clear that combining manual code review with automatic detection tools is the most effective way to identify vulnerabilities. However, I hope to develop more scripts in the future to make manual code review more convenient and potentially even automate it.

In the future, I plan to use python scripts to automatically filter out the false positives of FlawFinder and improve the visualization of its analysis results. I also plan to add a feature that allows users to click on a link to jump to the location of the code for manual inspection. Additionally, I hope to extract the issue and version update instructions from the repository and automatically map them to vulnerabilities, making it easier to track which vulnerabilities have been fixed.

6 ACKNOWLEDGEMENTS

I would like to express my gratitude to the professors and teaching assistants of the program analysis and security course for their assistance and guidance. Without their help, I would not have been able to complete this final course report. I would also like to thank Professor Wen Ming for carefully organizing the course tasks and topics. I hope that the program analysis and security course will continue to improve and be successful.

REFERENCES

[1] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 317–328. https://doi.org/10.1145/3238147.3238213

[2] shandianchengzi. 2022. *HUST CSE Program Analysis Course Project.* Retrieved January 1, 2023 from https://github.com/shandianchengzi/HUST_CSE_Program_Analysis_Course_Project