DOI: 10.15514/ISPRAS-2019-1(2)-1

Detecting errors in the Pandas software module using the Svace static code analyzer

[1] *M.A. Lapina,* ORCID: 0000-0001-8117-9142 <*mlapina@ncfu.ru*>
[1]*M.I. Khodakov* ORCID: 0009-0007-6848-7184< *mxkhodakov@yandex.ru* >
[1] *North Caucasus Federal University,*
1, Pushkina st., Stavropol, 355017, Russia

Abstract. The article deals with the urgent problem of software security at the early stage of its development. Special attention is paid to static code analysis, which is a key tool for detecting vulnerabilities at early stages of the software development life cycle. The article emphasises the importance of integrating static analysis tools into the development process in order to detect and eliminate vulnerabilities early. The methods of static analysers' error search are considered, as well as the main components of the Svace static analyser developed at the Institute of System Programming of the Russian Academy of Sciences. Classification of analyses used in the Svace static analyser is presented. Static analysis of source code in Python programming language is considered in detail. As a practical example the analysis of the Pandas 2.2.1 project performed with the help of Svace is given. The result was the detection of 241 vulnerabilities for 590709 lines of code, which shows a high density of warnings per million lines of code and confirms the effectiveness of static analysis in ensuring software security.

Keywords: static code analysis; static analyser; lexical analysers; lightweight analysers; abstract syntax tree; code fragment.

For citation: Lapina M.A., Khodakov M.I. Detecting errors in the Pandas software module using the Svace static code analyzer. *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2019. pp. 15-19 (in Russian). DOI: 10.15514/ISPRAS-2019-1(2)-1

1. Introduction

Nowadays, software is used everywhere. Statistics shows that as the number of programs increases, so does the number of vulnerabilities in them. That's why special attention is paid to software analysis.

At present there are two most popular methods of code analysis - dynamic analysis and static analysis. It is impossible to imagine software development without using static analysis tools. Static analysis is a type of analysis when the program is not executed but its whole code is analyzed [1]. This method is most often implemented at the initial stages of development, which helps to detect vulnerabilities earlier and eliminate them faster. Static analysis of source code can be performed in two ways - manually and with the help of special software tools.

In manual analysis of source code, checks such as code review or code inspection are performed. This approach is actively used because a human is able to detect defects in source code that software analyzers cannot yet.

Software analysis is performed using various analyzers. Many methods of static error search in programming have developed from the sphere of program compilation and operate with abstractions. Depending on the abstractions used, error search methods [2] can be divided into several groups:

- Lexical analyzers are designed to break down the source code of a program into individual tokens or tokens. They can be used to find only the simplest types of defects [3].
- Lightweight analyzers, also known as first-level analyzers, check the text for compliance with some grammar and build a parse tree (an abstract syntactic tree) by a linear sequence of tokens of this text, which is well suited for further processing and analysis of the text [4].
- More sophisticated parsers (Level 2 parsers), which use more complex algorithms and methods for parsing related to phases after syntactic analysis [5].

The disadvantage [6] of this method is that the analyzer may consider absolutely safe code fragments suspicious. Excessive suspiciousness leads to an increase of the false/true alarms ratio [7].

In this article we will consider the Svace static analyzer developed at ISP RAS and analyze the Pandas 2.2.1 project using it. This analyzer detects a large number of vulnerabilities in code, has a high level of true positives, fine-tuning and a large number of supported programming languages were the reasons for choosing this product.

2. Description of Svace

Svace is a tool developed at the Institute of System Programming of the Russian Academy of Sciences that performs static error search using several types of analyzers. It combines the key qualities of foreign analogs (Synopsis Coverity Static Analysis, Perforce Klocwork Static Code Analysis, Fortify Static Code Analyzer) with the unique use of open industrial compilers in order to maximize support for new programming language standards [8]. Supported languages are C, C++, C#, Java, Kotlin, Go, Python, Scala.

2.1 Svace analysis scheme

The schematic in (Fig. 1) shows the general analysis scheme of the Svace static analyzer [9].

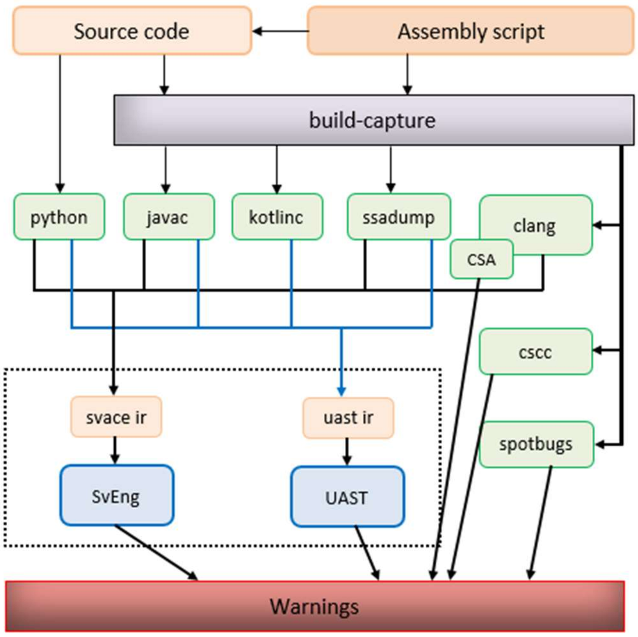

Fig.1. Schematic of the Svace static analyzer [9]

2.2 Svace components

The Svace static analyzer includes five components - SvEng, UAST, SpotBugs, CSA and cscc.

SvEng is the main component of Svace. It uses the following types of static analysis: pattern search in AST-trees, data flow analysis, interprocedural symbolic execution with state merging and using method summaries, static analysis of labeled data for security errors.

UAST - component in which errors are searched in a unified abstract syntax tree.

SpotBugs - component intended for analyzing programs in Java programming language.

CSA - intended for analyzing software in C/C++ languages.

cscc - is intended for analyzing programs in C# language. This component is based on the Roslyn compiler. For searching errors it uses the same types of analysis as SvEng.

2.3 Classification of analyses used at Svace

The analyses used in the Svace static analyzer [9] can be divided into the following groups:

- Analyses based on pattern search in the abstract syntax tree (ASD);
- Interprocedural analyses based on summaries;
- Statistical analyses;
- Special function analyses;
- Symbolic execution-based analyses;
- Labeled function analyses.

3. Analyzing a Python project with Svace

Analyzing the source code of a Python application does not require building the project because it is an interpreted language. Two components are involved in the analysis: SvEng and UAST.

Pandas was chosen as a project written in Python. 590709 lines of code were checked and 241 vulnerabilities were detected as a result of the analysis. The density of warnings per million code lines is 407.98. Table 1 shows the analysis results.

Table 1. Results of the analysis

Warning	Number of warnings
INVARIANT_RESULT	93
DIVISION_BY_ZERO.EX	23
BAD_COPY_PASTE	8
CATCH.NO_BODY	107
MUTABLE_DEFAULT_ARGUMENT	1
SIMILAR_BRANCHES	7
WRONG_ARGUMENTS_ORDER	2

Fig. 2 shows a code fragment where the INVARIANT_RESULT warning was detected in the project Pandas pandas/tests/window/moments/test_moments_consistency_ewm.py:

```python
    if adjust:
        count = 0
        for i in range(len(s)):
            if s.iat[i] == s.iat[i]:# INVARIANT_RESULT
                w.iat[i] = pow(1.0 / (1.0 - alpha), count)
                count += 1
            elif not ignore_na:
                count += 1
```

Fig. 2. Error INVARIANT_RESULT

The essence of this error is that the value of the expression does not depend on the values of its parts, or a part of the expression can be omitted without changing the result.

Fig. 3 shows the DIVISION_BY_ZERO.EX error in the project Pandas pandas/tests/io/parser/test_skiprows.py:

```python
def test_skip_rows_bad_callable(all_parsers):
    msg = "by zero"
    parser = all_parsers
    data = "a\n1\n2\n3\n4\n5"

    with pytest.raises(ZeroDivisionError, match=msg):
        parser.read_csv(StringIO(data), skiprows=lambda x: 1 / 0)
```

Fig. 3. Error DIVISION_BY_ZERO.EX

This error occurs whenever the program tries to divide an interval or a numeric value by 0.

The following types of errors were detected on the abstract syntax tree [10] using the component UAST (unified abstract syntax tree).

Fig. 4 shows a code fragment with BAD_COPY_PASTE error in the project Pandas pandas/tests/extension/base/constructors.py:

```python
        if hasattr(result._mgr, "blocks"): # Original code
            assert isinstance(result._mgr.blocks[0], EABackedBlock)
        assert result._mgr.array is data


        result2 = pd.Series(result)
        assert result2.dtype == data.dtype
        if hasattr(result._mgr, "blocks"): # Bad copy paste
            assert isinstance(result2._mgr.blocks[0], EABackedBlock)
```

Fig. 4. Error BAD_COPY_PASTE

An error occurs if the code has been copied and reproduced, but not all necessary changes have been made.

Fig. 5 shows the CATCH.NO_BODY error in the project Pandas pandas/tests/plotting/test_converter.py:

```
try:
    from pandas.plotting._matplotlib import converter
except ImportError: # CATCH.NO_BODY


    pass
```

Fig. 5. Error CATCH.NO_BODY

This error occurs if an exception was caught but not handled by the corresponding except block.

Fig. 6 shows a code fragment with the MUTABLE_DEFAULT_ARGUMENT error in the project Pandas pandas/io/formats/style_render.py:

```
def __init__(
    self,
    css_props: CSSProperties = [  #MUTABLE_DEFAULT_ARGUMENT
        ("visibility", "hidden"),
        ("position", "absolute"),
        ("z-index", 1),
        ("background-color", "black"),
        ("color", "white"),
        ("transform", "translate(-20px, -20px)"),
    ],
```

Fig. 6. Error MUTABLE_DEFAULT_ARGUMENT

The error occurs if we use some modifiable object (list, dictionary) as a default value for a function argument. If we change the value of the argument inside the function, we will have to change the original value because they both refer to the same object, in this case it is the css_props argument. This can lead to unexpected results and errors in your code.

Fig. 7 shows a code fragment with SIMILAR_BRANCHES_ARGUMENT error in the project Pandas pandas/tests/indexes/datetimes/test_date_range.py:

```
elif inclusive_endpoints == "both":
    expected_range = both_range[:] # First branch
else:
    expected_range = both_range[:] # Second branch
```

Fig. 7. Error SIMILAR_BRANCHES

This error occurs when executing the same instructions regardless of the condition.

Fig. 8 shows a code fragment with WRONG_ARGUMENTS_ORDER ARGUMENT error in the project Pandas pandas/_testing/__init__.py:

```
if isinstance(left, np.ndarray) and isinstance(right, np.ndarray):
    return np.shares_memory(left, right)
elif isinstance(left, np.ndarray):

    return shares_memory(right, left) # WRONG_ARGUMENTS_ORDER
```

Fig. 8. Error WRONG_ARGUMENTS_ORDER

The error occurs when method arguments are passed in the wrong order. In this example, left and right are mixed up in the wrong order.

4. Classification of errors

Errors in software can be classified by the degree of their influence on the program operation [11]. Table 2 presents the table of error classification and their description.

Table 2. Classification of errors and their description

Error class	Description
Critical errors	This class of errors leads to an emergency situation that makes the program operation impossible.
Major errors	Errors that lead the program to unexpected results.
Minor errors	Errors that reduce the efficiency of the program and its performance.

Errors obtained as a result of static analysis were classified by the level of their influence on the program code.Table 3 presents the error classification table.

Table 3. Classification of errors

Class	Error name
Critical error	DIVISION_BY_ZERO.EX
Major error	MUTABLE_DEFAULT_ARGUMENT
	WRONG_ARGUMENTS_ORDER
Minor error	INVARIANT_RESULT
	BAD_COPY_PASTE
	SIMILAR_BRANCHES
	CATCH.NO_BODY

The DIVISION_BY_ZERO.EX error belongs to the "Critical error" class because division by zero in the program code may cause unpredictable behavior of the command and eventually lead to its crash. Fig. 9 shows the scenario when this error occurs.

```
def divide_numbers(num1, num2):
    return num1 / num2

def main():
    values = [10,0]
    for i in range(len(values) - 1):
        print(f"division {values[i]} by {values[i+1]}")
        outcome = divide_numbers(values[i],values[i+1])
        print(f"Result: {outcome}")

        if __name__ == "__main__":
            main()
```

Fig. 9. *Error scenario DIVISION_BY_ZERO.EX*

The MUTABLE_DEFAULT_ARGUMENT error belongs to the "Major error" class because it can cause unexpected results when using a variable data type as a default value for an argument. Fig.10 shows a scenario where this error occurs.

```
def item(item, item_list=[]):
    item_list.append(item)
    return item_list


print(item('apple'))
print(item('banana'))
```

Fig.10. *Error scenario MUTABLE_DEFAULT_ARGUMENT*

In this code, the item function uses the variable data type (list) as the default value for the item_list argument. As a result, each time the add_item function is called, items are added to the same item_list instead of a new list. This is the MUTABLE_DEFAULT_ARGUMENT error.

The WRONG_ARGUMENTS_ORDER error is a "Major error" because if the order of arguments in a function call is incorrect, it can lead to unexpected program behavior. Fig. 11 shows the scenario of this error.

```
def main(x, y):
    return f"{y}, {x}!"

print(main("x", "y"))
```

Fig.11. *Error scenario WRONG_ARGUMENTS_ORDER*

In this example, the main function expects the first argument to be "x" and the second argument to be "y". However, when the function is called, these arguments are specified in reverse order, which causes an error.

The errors INVARIANT_RESULT, BAD_COPY_PASTE, SIMILAR_BRANCHES, CATCH.NO_BODY are not serious and belong to the "Minor error" class, but they require corrections to prevent unexpected program behavior.

Fig. 12 shows the SIMILAR_BRANCHES error scenario.

```
x = 10

if x > 5:
    y = x * 2
else:
    y = x * 2
```

Fig. 12. . Error scenario SIMILAR_BRANCHES

In this example, the if and else branches contain the same instructions y = x*2, which may affect the efficiency of the program.

One of the examples of the BAD_COPY_PASTE error is shown in Fig. 13.

```
def calculate_sum(a, b):
    return a + b


def calculate_difference(a, b):
    return a + b
```

Fig.13. Error scenario BAD_COPY_PASTE

In this example, the calculate difference function should calculate the difference between variables a and b, but due to code copying, an error was made that may cause the program to work incorrectly.

The CATCH.NO_BODY error scenario is shown in Fig. 14.

```
try:
    x = 1 / 0
except ZeroDivisionError:

    pass
```

Fig.14. Error scenario CATCH.NO_BODY

In this example, the exception is not handled in the except block, which may cause the error to be ignored and the program to continue execution without handling the error.

Fig. 15 shows the INVARIANT_RESULT error scenario.

```
def calculate(x, y, z):


    result1 = x * 0 + y * 0 + z * 0

    result2 = x + y - y + z

    result3 = x + y + z - z
```

Fig.*15. Error scenario INVARIANT_RESULT*

In this example, the value of result1 does not depend on the variables x, y, z, because multiplication by 0 will always result in 0. The value of result2 will always be x + z because y - y is 0, so this part of the expression can be omitted without changing the result. Similarly with the value of result3, it will always be equal to x + y because z - z is 0.

5. Conclusion

As it was shown in the article, static analysis tools help detect errors at the initial stages of application development. Different static analyzers use different algorithms for detecting vulnerabilities in application source code. Svace static analyzer allows you to analyze program code in different languages. We also analyzed the project in Python, Pandas 2.2.1. In the process of analysis we found errors of different classes. Classification of errors by their level of influence on the program code, their full description and examples of scenarios when these errors can occur in the program code were carried out. The results of the research may be useful for improving the quality and reliability of the product.

References

[1]. Chernov D. Code analysis: problems, solutions, perspectives. 2022. URL: https://www.tadviser.ru/index.php/Статья:Анализ_кода:_проблемы,_решения,_перспективы#:~:text=У%20метода%20есть%20два%20недостатка,программы%2C%20который%20не%20всегда%20доступен, accessed 20.03.2024

[2]. Borodin A.E., Belevantsev A.A. Static analyzer Svace as a collection of analyzers of different levels of complexity. Proceedings of ISP RAS, vol. 27, issue 6, 2015, pp. 111-134. (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

[3]. Zaboleva-Zotova A.V., Orlova Y.A. MODELING OF LEXICAL ANALYSIS OF TECHNICAL TASK TEXT. URL: https://www.elibrary.ru/item.asp?id=9506673, accessed 21.03.2024

[4]. Mavchun E.V. Comparison of algorithms of tabular bottom-up syntactic analysis. URL: https://oops.math.spbu.ru/SE/dip loma/2015/s/544-Mavchun-report.pdf, accessed 24.03.2024

[5]. A. P. Syzranov, A. I. Nenakhov, A. Y. Bolotov, A. J. Osipov. Development of criteria for evaluation of means of automation of certification testing on the level of control of the absence of undeclared capabilities. URL: https://elibrary.ru/item.asp?id=47856981, accessed 24.03.2024

[6]. Fadeev S.G. TECHNOLOGY OF STATIC ANALYSIS OF SOURCE CODE. URL: https://apni.ru/media/Sbornik-6-3.pdf#page=131, accessed 20.03.2024

[7]. Kuchin I.Yu. Review of existing methods of program code analysis. URL: https://cyberleninka.ru/article/n/obzor-suschestvuyuschih-metodov-analiza-programmnogo-koda/viewe, accessed 21.03.2024

[8]. Svace static analyzer. URL: https://www.ispras.ru/technologies/svace/, accessed 24.03.2024

[9]. Svace components. URL: https://svace.pages.ispras.ru/svace-website/2023/10/04/analysis-types.html, accessed 24.03.2024

[10]. Afanasyev V.O., Borodin A.E., Vikhlyantsev K.I., Belevantsev A.A. Static analysis based on generalized abstract syntactic tree. Proceedings of ISP RAS, vol. 35, issue 6, 2023, pp. 103-120. (in Russian). DOI: 10.15514/ISPRAS–2023–35(6)–6.

[11]. V.V. Bykova, G.E. Glukhov, A.N. Sharypov, P.E. Chernikov, S.V. Koval, A.Yu. Konkov. PROBLEMS OF VULNERABILITY OF INFORMATION SYSTEMS OF AVIATION INDUSTRY ENTERPRISES: ANALYSIS AND CLASSIFICATION OF ERRORS. URL: https://mlgvs.ru/files/iac/art-niiga-2019-27.pdf, accessed 01.04.2024

Information about authors

Lapina Maria Anatolyevna – Candidate of Physical and Mathematical Sciences, Associate Professor of the Department of Information Security of Automated Systems of the North Caucasus Federal University.

Khodakov Maxim Ivanovich – student of the Department of Information Security of Automated Systems of the North Caucasus Federal University.