Extending Rule Set for Static Code Analysis in .NET Platform

Jonas Čeponis, Algimantas Venčkauskas, Lina Čeponienė, Andrius Zonys

Kaunas University of Technology, Faculty of Informatics,
Studentų 50, LT-51368 Kaunas, Lithuania

e-mail: jonas.ceponis@ktu.lt, algimantas.venckauskas@ktu.lt, lina.ceponiene@ktu.lt, a.zonys@gmail.com

crossref http://dx.doi.org/10.5755/j01.itc.45.1.9341

Abstract. This paper focuses on static code analysis tools for .NET platform. Static code analysis tools typically use a certain set of rules. In this paper, we propose to implement four rules, which we consider important from our practical experience of software development. We analyze the existing popular static analysis tools for .NET platform in order to determine whether they have the rules equivalent to our new rules. We select an open-source tool Gendarme for the implementation of these rules. We also investigate existing Gendarme rules and discover that some of them could be improved. Therefore, we propose and implement improvements for four existing Gendarme rules. In order to evaluate the improvements made in Gendarme rule set in a real-life environment, the source code of five open-source programs from sourceforge.net is tested using new and improved rules. Results indicate that new and improved Gendarme rules enable detection of more errors and can increase the quality of source code.

Keywords: static code analysis; .NET; code analysis rules; code defects.

1. Introduction

Source code analysis methods can be divided into three main categories. The first category encompasses manual code review techniques [11]. The second category comprises static code analysis methods [2]. During static code analysis, code review is performed automatically without executing the program [6], but some errors are difficult to find without execution [24]. Therefore, the third category – dynamic code analysis methods can be used. Dynamic code analysis performs code review automatically during program execution [1].

During the static code analysis, testing is performed using the rules for detecting errors in source code. Rules are usually designed for each error or for the group of errors [9] [14] [27]. Static code analysis tools can detect code defects and report their location [4] [8] [21]. Tools can report about the error found and often suggest the correction for it [15] [26]. Some of them can create detailed reports about any problems found in code. Tools should have an option of ignoring found defects as these may be false errors. Static code analysis tools often detect false errors, and tools are being improved in order to reduce the false alarm rate [3].

Static code analysis tools come in different types, targeted to various platforms [10] [12] [18] and use various methods for analysis [20]. Some of these tools analyze code before compilation, and others apply code analysis after compilation [16]. The compiled and optimized binary code may differ from the initial one. Static analysis tools read the code and create the abstract model. This model is used for searching for code defects according to the list of errors and their corresponding rules. Some tools can also perform data flow analysis in order to predict values of variables in certain code locations. Human errors are somewhat predictable, but developers of analysis tools are not able to define all possible errors. As a result, some of the tools enable programmers themselves to create rules for error detection.

The terms “code defect”, “error”, “vulnerability” are used interchangeably in this paper. By all these terms we mean the issue in the code, which was not detected by the compiler and did not prevent compilation of this code. But these undetected issues can cause problems during execution of the applications or during further improvement and reuse of source code [5] [19].

In this paper, we analyze static code analysis tools for .NET platform and propose the extension of the rule set for the open-source tool Gendarme [22]. Both the new rules and improvements of existing rules are proposed in our work. The proposed extensions are implemented and tested in practice using available code of several open-source programs.

The rest of the paper is organized as follows. In Section 2, related work is analyzed. In Section 3, tools for static code analysis in .NET are investigated. The
set of new rules, based on our practical experience in software development is presented and the tools are examined in order to find out whether they already have these rules implemented. Section 4 presents the implementation of four new rules in Gendarme tool. In Section 5, improvements of existing Gendarme rules are presented. Section 6 describes the results of experiment during which the new rules and the rule improvements were tested on the source code of selected open-source applications. Section 7 ends the paper with conclusions.

2. Related work

There are various techniques for improving static code analysis and extending existing static code analysis tools. Some of these techniques are focused on the optimization of static analysis process, e.g. a sparse evaluation [17]. The reuse of static code analysis rules can also aid in extending the rule set of static analysis tool, therefore development of rule repository is proposed in [25]. We should also have in mind that some static code analysis rules can be specific for a certain project or for the certain team of programmers. There are researches on mining the project specific rules and expending the rule set of the certain static code analysis tool with these discovered rules. The discovered rules are transformed into XML rule specifications and these XML specifications are used for generating tool specific checkers [27].

On the other hand, the developers of analysis tools are not able to specify rules for detecting all possible errors and, therefore, static analysis tools should have the options of extending the ruleset with definitions of custom rules. Custom rules for static code analysis can be defined using domain specific language [23]. In [9], rules are defined in high-level tool independent language, but the grammar of the language is not presented. Dalci and Steven [9] also proposed the custom rule development process, which is test driven and requires development of test cases for each rule.

Another approach for specifying custom rules is using a programing language used by static analysis tool [7][13]. This approach has some drawbacks, as it concentrates on specifics of implementing the rules for the certain tool, but it is also the most straightforward approach for developing and testing a custom rule.

In our work, we are focusing on practical implementation of custom rules, therefore we implemented the rules in a programming language used by a static analysis tool. In this paper, each rule is explained and examples of correct and incorrect code for each rule are presented.

3. Tools and new rules for static analysis in .NET

In this paper, we investigate static source code analysis tools for .NET platform. The choice of tools for this platform is certainly broad – both commercial and open-source. One of the main tools for .NET application development is Visual Studio IDE developed by Microsoft. Therefore, most of .NET static analysis tools can be integrated into this IDE, and are convenient to use for developers.

Commercial static code analysis tool CodeIt.Right is developed by submain. It supports C# and VB.NET programing languages. This tool combines static code analysis and automatic refactoring for supporting best coding practices. Another commercial analysis tool ReShaper is created by JetBrains for Microsoft Visual Studio. It is equipped with a set of features for .NET developers working with C#, VB.NET, ASP.NET, XML, XAML, and build scripts. ReShaper analyzes errors and warnings in code and highlights them in the editor. For most errors it instantly offers solution for the problem. Commercial analysis tool CodeRush created by Developer Express is implemented as plug-in for Microsoft Visual Studio. It finds issues in code, underlines them and puts indicators on the corresponding code lines. Another static analysis tool, NDepend, simplifies managing a complex .NET code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of code. Free tool StyleCop analyzes C# source code using a set of style and consistency rules. It can be run from inside of Microsoft Visual Studio or integrated into an MSBuild project. This tool is intended for ensuring consistent style and formatting of source code. Commercial analysis tool Parasoft dotTEST is developed as a plugin for Visual Studio and has static analysis, unit testing, and code review features. dotTEST works with languages for Microsoft .NET Framework and .NET Compact Framework, including C#, VB.NET, ASP.NET and Managed C++.

In our practical experience of software development, we have developed the list of errors which we consider important. Detecting these types of errors (or code defects) in static code analysis tool would be very helpful for the programmer:

- finding names of variables or methods which differ only in letter case;
- checking whether method always returns one and the same value from different code locations;
- checking whether the variable is not further used after it is determined as being null;
- detecting conditions which cause infinite loops in while and do-while.

A more detailed description of these error types is presented in Sections 4.1-4.4.

We have analyzed the above mentioned commercial and free tools for .NET platform in order to determine whether they detect these types of code defects. The static analysis tools were integrated into Visual Studio 2012 Ultimate and tested with simple code examples ( Fragments of these code examples are presented in Section 4. They were complemented for a successful compilation). Testing results are presented in Table 1.
## Table 1. Detection of code defects in .NET static analysis tools

<table>
<thead>
<tr>
<th>Analyzed tools</th>
<th>Code defects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>names differ only in letter case</td>
</tr>
<tr>
<td>ReShaper (version 8.2)</td>
<td>+/-</td>
</tr>
<tr>
<td>CodeRush (version 14.1)</td>
<td>-</td>
</tr>
<tr>
<td>StyleCop (version 4.7)</td>
<td>-</td>
</tr>
<tr>
<td>CodeIt.Right (version 2.2)</td>
<td>+/-</td>
</tr>
<tr>
<td>NDepend (version 4.1)</td>
<td>-</td>
</tr>
<tr>
<td>Parasoft dotTest (version 9.5)</td>
<td>-</td>
</tr>
</tbody>
</table>

As we can see in Table 1, ReShaper detected an error if a variable was used after its value was determined as null. This tool also found infinite loops, but marked them as unreachable code. ReShaper and CodeIt.Right tools discovered method names that differ only in upper and lower case letters, but the discovery was not always correct: after changing some lowercase letters to uppercase, the tools did not detect this error. The code issue when method always returns the same value from different code locations was not discovered by any of the tested static analysis tools.

Our research showed that detecting code defects, which we consider important, is not implemented (or is just partially implemented) in any of the analyzed tools. Therefore we have decided to develop the rules for detecting and reporting these code defects. We made a decision to choose an open-source tool for implementation of our proposed error detection rules in order to have the possibility of improving the functionality of the tool, if it would be required. One of our analyzed tools, StyleCop, is an open-source tool, but it focuses more on code style issues. We needed a tool which would focus on deeper code analysis. Therefore we chose an open-source tool Gendarme [22]. Gendarme is an expandable, rules-based tool designed to detect code errors in .NET applications and libraries. Gendarme analyzes applications and libraries compiled in ECMA CIL format and looks for the most common programming errors and mistakes, which normally cannot be detected by the compiler. Gendarme code analysis tool uses Cecil library. The rules for code analysis in Gendarme are used by runners. A runner is a program module responsible for the load of rules, analysis of code and error reporting. Gendarme (version 2.10) has two different runners. One is based on a command-line interface, the other has graphical interface. Command-line runner can output results to the command line, XML or HTML files. A graphical interface runner enables choosing step by step modules (called assembly), rules, and settings. Gendarme static code analysis tool is modular and all rules are divided into certain groups. Command-line runner uses all rules that are specified in the configuration file rules.xml. This file can be adjusted according to user’s needs. Gendarme has user-friendly configuration possibilities and a clear representation of analysis results.

During analysis of existing rules in Gendarme tool, we also found several rules which are intended for detecting severe code defects but have significant drawbacks and should be improved to ensure better error detection. Therefore we have not only developed four new rules for detecting defects, but also proposed the improvements for four existing Gendarme rules.

### 4. Implementation of new static code analysis rules in Gendarme

In our work, four new rules are proposed and implemented in static code analysis tool Gendarme. The main purpose for implementing these rules was detection of excessive, illogical code, which existing Gendarme v.2.10 rules cannot detect. The other nonetheless important goal is to simplify the code and thus to improve its readability and help detect more security vulnerabilities. In order to improve the code inspection, four new rules have been implemented:

- **AvoidCaseSensitiveNamesRule** – the rule for finding names that differ only in uppercase or lowercase.
- **AvoidReturnSameValueRule** – the rule for finding possibly incorrect return value.
- **AvoidUsingNullAfterNullityCheckRule** – the rule for avoiding unexpected termination of the program due to usage of variable with null value.
- **AvoidInfiniteLoopRule** – the rule for finding infinite loops.

The next Sections (4.1–4.4) describe these rules in detail. Examples of incorrect and correct code are also presented for each rule.
In programming, a naming convention is often used. It is a set of rules for choosing the naming of identifiers which denote variables, types, methods, etc. The main goal for using a naming convention is to increase readability and understandability of the code and to enhance its appearance. Some of the naming conventions define the rules for using uppercase or lowercase letters. Conventions also attach a well-defined interpretation based on letter case. The most popular programming languages – C, Java, Objective-C, C++, & C# (based on Tiobe index - www.tiobe.com) – are case-sensitive for their identifiers. Microsoft recommends using UpperCamelCase (a.k.a. "Pascal Style") for most identifiers in .NET. lowerCamelCase is recommended for parameters and variables and is a shared convention for the .NET languages. Microsoft further recommends that no type prefix hints (also known as Hungarian notation) are used.

While using case-sensitive naming conventions, the situation can occur when the names of several variables or methods differ only in letter case. For example, two different methods, performing different actions, can be named ConvertToXMLFormat() and ConvertToXMLEFormat(). The programmer can use the inappropriate method just because of letter case confusion in the method name.

Static code analysis tool should detect all identifier names which differ only by letter case and report to programmer. In order to resolve the case-sensitive naming confusion problem we implemented the new rule (called AvoidCaseSensitiveNamesRule) in Gendarme.Rules.Naming category. This rule detects methods and variable names that differ only in letter case and reports an error. Names of variables are analyzed separately from the names of methods, i.e. variable and method can have the same name and this will not be treated as an error. An improper source code example where two method names differ only in letters case is presented in Fig. 1 a). A possible code correction example is presented in Fig. 1 b). AvoidCaseSensitiveNamesRule rule algorithm first checks all class variable names. Variable names are converted to lowercase, and one by one appended to a new list. If the name already exists in the list, an error is recorded. The next step of the algorithm is analysis of method names. In this step, the identical method names are ignored because they have different parameters. Variables in methods are checked in the same manner as the class variables.

4.2. AvoidReturnSameValueRule

A return statement causes leaving current method and resuming at the point in the code immediately after where the method was called. Return statement defines a return value, which is passed to the code that called the function. Return of incorrect value is a common mistake. One of the possible ways of checking if the return value is correct is determining whether the values returned from different locations in the method differ from each other. If the method always returns one and the same value from different code locations, then what is the point of such return? If the value must be returned, then constant variable can be created and returned as a value through the property, or simply as a function return value. This should facilitate the readability of the program and help to avoid distraction errors. Our proposed new rule AvoidReturnSameValueRule checks whether the method can return different values.

An improper source code example where method DoSomething always returns value false from different code locations is presented in Fig. 2a). In such situation the tool will report an error. An example of correct code is presented in Fig. 2 b).

The rule implementation compares the first found return value with all further return values in the method. If all returned values are identical, an error is recorded. If the algorithm detects that the return value is another method return value (eg. x.Count ()), or an expression of type boolean (eg. return a > b), it is treated as the return of different values.

4.3. AvoidUsingNullAfterNullityCheckRule

Before using a variable of a public method it is common to check whether its value is not null. The usage of variable whose value is null can cause unexpected termination of a program. In the Common Weakness Enumeration (CWE™) list, there are several weaknesses associated with checking variables whether they have a value or not. But even if the value is checked in the program, situations can occur (e.g. in complex if statements) where variables with null
Extending Rule Set for Static Code Analysis in .NET Platform

```csharp
public bool DoSomething (int index)
{
    if (index < 0)
        return false;
    DoSomethingMore (index);
    return false;
}
```

```csharp
public bool DoSomething (int index)
{
    if (index < 0)
        return false;
    DoSomethingMore (index);
    return true;
}
```

Figure 2. Incorrect and correct code examples for AvoidReturnSameValueRule rule

```csharp
public void Method (string param)
{
    if (param != null || param.Length > 3)
        Console.WriteLine ("Acceptable.");
    Console.WriteLine ("NOT acceptable.");
}
```

```csharp
public void Method (string param)
{
    if (param != null && param.Length > 3)
        Console.WriteLine ("Acceptable.");
    Console.WriteLine ("NOT acceptable.");
}
```

Figure 3. Incorrect and correct code examples for AvoidUsingNullAfterNullityCheckRule rule

```csharp
public void AvoidInfiniteLoop ()
{
    int a = 0;
    while (true)
    {
        DoSomething (a++);
        if (a > 1000 || !DoSomething (a++))
            break;
    }
}
```

```csharp
public void AvoidInfiniteLoop ()
{
    int a = 0;
    while (true)
    {
        if (a > 1000 || !DoSomething (a++))
            break;
    }
}
```

Figure 4. Incorrect and correct code examples for AvoidInfiniteLoopRule rule

```csharp
public void ExampleMethod (string param)
{
    string paramCopy = string.Copy (param);
    if (param.Length > 10)
        ...
    ...
}
```

```csharp
public void ExampleMethod (string param)
{
    string paramCopy = string.Copy (param);
    if (param != null && param.Length > 10)
        ...
    ...
}
```

Figure 5. Incorrect and correct code examples for CheckParametersNullityInVisibleMethodsRule rule

value are used. These errors often occur during correction or overwriting of the complex code. In order to minimize the chances of using a variable with null value, we designed the rule AvoidUsingNullAfterNullityCheckRule. This rule checks whether the detected variable, whose value is null, is no longer used.

In the code example in Fig. 3 a), if param value is null (line 3), program execution will end abnormally while executing if statement. Verification of the first part of the condition in if statement (param != null) will be executed normally. However, the second part of the condition in if statement also has to be verified (as there is an operator ||) and this action will cause unexpected termination of the program. Our new rule will detect such situation and report an error. A possible correction of such error is presented in Fig. 3 b): if operator || is replaced by operator && (line 3), the program will work correctly in case of null value of variable param.

The proposed rule AvoidUsingNullAfterNullityCheckRule is incorporated in Gendarme.Rules.Design category. The algorithm of this rule is searching for conditions where the variable or parameter is checked for null value. If such condition is found, the code is further inspected till the end of method or until the variable is assigned with a new value. During inspection the rule checks whether this null variable is used or whether there is an attempt to call the methods of the variable.

4.4. AvoidInfiniteLoopRule

Infinite loop is a common type of programming errors and may cause the meaningless activity of the program. However, determining whether the program
is stuck in the infinite loop or is performing necessary calculations is often an unsolvable problem. It is impossible to detect all kinds of infinite loops fully automatically.

Static analysis tool integrated into Microsoft Visual Studio Ultimate 2012 detects infinite loops in ill-defined for loop. In order to improve detection of infinite loops we have proposed the rule for detecting ill-defined while or do-while loops. In the code example presented in Fig. 4 a) Gendarme tool did not detect an infinite loop, although it is clearly visible. Sometimes while (true) or similar type of loop can be written consciously, but in this case the conditions and actions for terminating the loop must be defined. We have proposed the rule AvoidInfiniteLoopRule which checks the presence of conditions for terminating the loop.

It will detect an infinite loop in code example presented in Fig. 4 a). Possible infinite loop correction is presented in Fig. 4 b).

AvoidInfiniteLoopRule rule first looks for a loop in program code. Upon detection of the loop all variables used in the loop condition are analyzed trying to detect whether the condition variable can acquire new value and whether there is a break or return instructions. If these operations are not detected, the error is reported.

5. Improvements of existing Gendarme rules

The rules in Gendarme (version 2.10.8) are divided in 17 categories (for instance Concurrency, Design, Security, etc.) and cover various areas of code defects. During analysis of existing rules, we have discovered that some of them can be improved, thus ensuring more thorough code analysis and error detection. Therefore we proposed improvements for four Gendarme rules:

- **CheckParametersNullityInVisibleMethodsRule**
- **AvoidUnusedPrivateFieldsRule**
- **RemoveUnusedLocalVariablesRule**
- **ArrayFieldsShouldNotBeReadOnlyRule**

The next Sections (5.1–5.4) describe improvements of these rules in detail. Examples of incorrect code in which existing rules do not detect errors are presented for each rule. Improved Gendarme rules detect the errors in presented incorrect code examples. The examples of possible corrections for code defects are also presented for each improved rule.

5.1. CheckParametersNullityInVisibleMethodsRule

CheckParametersNullityInVisibleMethodsRule belongs to Gendarme.Rules.Correctness category. The rule states that visible parameters of methods must be checked to ensure that their value is not null before using them. Original Gendarme rule did not check nullity for a certain parameter if any method was called using that parameter. Improved rule does not treat this action as null value test.

An improper source code example, when the original Gendarme rule will not detect possible null value and program execution may be terminated abnormally due to NullReferenceException is presented in Fig. 5 a). Our improved rule will suggest correcting the code by adding null value check in line 4. The correct code example is presented in Fig. 5 b).

However, one problem still remains: the algorithm only detects whether the parameter is checked for being equal or not equal to null, but does not analyze how this parameter is further used. Therefore we created the new rule AvoidUsingNullAfterNullityCheckRule which was described in Section 3.3.

```
1 public class Example
2 {
3     private string m_unusedV;
4     private const string m_unusedC;
5     public void ExampleMethod ()
6     {
7         m_unusedV = "Unused variable";
8         m_unusedC = "Unused const";
9         DoSomething ();
10     }
11 }
```

a)

```
1 public class Example
2 {
3     private string m_unusedV;
4     private const string m_unusedC;
5     public void ExampleMethod ()
6     {
7         m_unusedV = "Unused variable";
8         m_unusedC = "Unused const";
9         DoSomething ();
10     }
11 }
```

b)

Figure 6. Incorrect and correct code examples for AvoidUnusedPrivateFieldsRule rule

```
1 public void ExampleMethod ()
2 {
3     String unused = "Unused variable";
4     DoSomething ();
5 }
```

a)

```
1 public void ExampleMethod ()
2 {
3     String unused = "Unused variable";
4     DoSomething ();
5 }
```

b)

Figure 7. Incorrect and correct code examples for RemoveUnusedLocalVariablesRule rule
5.2. AvoidUnusedPrivateFieldsRule

This rule belongs to the Gendarme.Rules.Performance category and is used for detecting unused private class variables. If the private variable was assigned with a value but was not further used, original Gendarme rule did not treat such situation as a code defect. Our improved rule will suggest removing such private variable and its assignment operations. Our rule will not detect an error only if the value of private variable is used. We also improved the rule to detect unused private constants.

An improper code example when original Gendarme rule did not detect any errors is presented in Fig. 6 a). In order to improve the efficiency of the program, unused variables in lines 3 and 4 should be removed from the code. A corrected code example is presented in Fig. 6 b).

Our improved rule checks all class operations until detects the one which reads the variable. Previously only the simple search was performed, without considering whether the detected usage of variable is value assignment or reading. Improvement of this rule enables decreasing required amount of memory and increasing performance.

5.3. RemoveUnusedLocalVariablesRule

This rule belongs to the Gendarme.Rules.Performance category. The rule is designed for detecting unused variables in the method. But this rule only searches for mentioning of the variable in method body. Improved rule not only searches for variable usage but also analyzes how it is used. The new rule detects an error if the method variable is never read in a method. An improper code example when original Gendarme rule did not detect any errors is presented in Fig. 7 a). In order to increase efficiency of the program, unused variables should be removed from the code. The improved rule checks all method operations until detects reading of the variable. If variable is not read, it is removed from the code. The corrected code is presented in Fig. 7 b).

The main purpose of using this rule is increasing efficiency of the program code. This is achieved by removing unused parts of the code – in this case, these unused parts are local variable definitions and value assignments to them.

5.4. ArrayFieldsShouldNotBeReadOnlyRule

The rule belongs to the Gendarme.Rules.Security category. This rule detects the arrays that are declared as public readonly and reports an error. If array is declared as public readonly, it cannot be changed, but values in the array can be changed. Such a situation may compromise security of the program. Our improved rule also detects declared public readonly Collection because it may be affected by the same security flaw. An improper code example when analysis tool does not detect errors is presented in Fig. 8 a).

The improved rule checks for variables in all classes. First, it checks if a variable is read-only and whether it is public. Afterwards it determines whether it is an array, a list or a collection. If variable meets both criteria, an error is reported. For correcting such error, the variable should become private and the methods for getting its values should be created. A possible code correction is presented in Fig. 8 b).

<table>
<thead>
<tr>
<th>1</th>
<th>public class Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{</td>
</tr>
<tr>
<td>3</td>
<td>public readonly Collection&lt;int&gt;</td>
</tr>
<tr>
<td>4</td>
<td>roColl = new Collection&lt;int&gt;()</td>
</tr>
<tr>
<td>5</td>
<td>public readonly ICollection&lt;int&gt;</td>
</tr>
<tr>
<td>6</td>
<td>roIColl = new List&lt;int&gt;()</td>
</tr>
<tr>
<td>7</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 8. Incorrect and correct code examples for ArrayFieldsShouldNotBeReadOnlyRule rule

<table>
<thead>
<tr>
<th>1</th>
<th>public class Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{</td>
</tr>
<tr>
<td>3</td>
<td>private readonly Collection&lt;int&gt;</td>
</tr>
<tr>
<td>4</td>
<td>roColl = new Collection&lt;int&gt;()</td>
</tr>
<tr>
<td>5</td>
<td>private readonly ICollection&lt;int&gt;</td>
</tr>
<tr>
<td>6</td>
<td>roIColl = new List&lt;int&gt;()</td>
</tr>
<tr>
<td>7</td>
<td>public int[] GetCollectionAsArray()</td>
</tr>
<tr>
<td>8</td>
<td>{</td>
</tr>
<tr>
<td>9</td>
<td>int[] m = new int[roColl.Count];</td>
</tr>
<tr>
<td>10</td>
<td>roColl.CopyTo(m, 0);</td>
</tr>
<tr>
<td>11</td>
<td>return m;</td>
</tr>
<tr>
<td>12</td>
<td>}</td>
</tr>
<tr>
<td>13</td>
<td>public int[] GetICollectionAsArray()</td>
</tr>
<tr>
<td>14</td>
<td>{</td>
</tr>
<tr>
<td>15</td>
<td>int[] m = new int[roIColl.Count];</td>
</tr>
<tr>
<td>16</td>
<td>roIColl.CopyTo(m, 0);</td>
</tr>
<tr>
<td>17</td>
<td>return m;</td>
</tr>
<tr>
<td>18</td>
<td>}</td>
</tr>
</tbody>
</table>

a) b)
6. Practical evaluation of new and improved Gendarme rules

In order to determine whether our new and improved rules are valuable in software development, we have decided to apply static code analysis using these rules to several popular open-source applications. Analysis of these applications demonstrates whether programmers are prone to make errors which are detected by our new and improved rules. Applications for testing were selected from sourceforge.net page.

We have filtered English applications written in C# programming language for Windows operating systems in production/stable state. We sorted filtering results by popularity and selected applications having approximately from 15 to 50 thousand lines of code. We selected ZedGraph, PDFsharp, NetworkMiner, WatiN and ASProxy applications.

All selected applications were analyzed with Gendarme tool using improved rules and their original versions. The results of analysis are presented in Table 2. The biggest number of errors was detected by CheckParametersNullityInVisibleMethodsRule. The overall number of errors detected by original rule is 652, and the improved rule detected even more – 855 errors. The benefit of improvement of RemoveUnusedLocalVariablesRule rule is clearly visible in analysis results – original rule did not detect any errors and the improved rule detected 137 total errors in all analyzed applications. The least number of errors (7) was detected by ArrayFieldsShouldNotBeReadOnlyRule rule.

The improved version of this rule detected only 3 more errors. The selected applications were also analyzed with Gendarme tool using our new rules. The results of analysis are presented in Table 3.

The biggest number of errors was detected by AvoidCaseSensitiveNamesRule rule in application PDFsharp. Such errors can cause serious problems if this application would be improved by other developers: they may be confused with variable or method names which differ only in letter case. The rule AvoidReturnSameValueRule detected errors in four applications, with total 13 errors.

The rule AvoidInfiniteLoopRule found only 2 errors in one of the applications (NetworkMiner), but these errors should be ignored because infinite loop was created intentionally (infinite loop in the application was used in separate threads for processing various incoming messages). From the point of view of software security, most problems can be caused by errors that are detected by AvoidUsingNullAfterNullityCheckRule rule. This rule found 10 errors in analyzed applications.

During static code analysis of selected applications, the biggest number of code defects was detected using improved versions of four Gendarme rules. The number of errors found by the new rules implemented in Gendarme was not that big. However, although the numbers are smaller, the errors still exist in various applications and can be successfully detected using our new rules.

<table>
<thead>
<tr>
<th>Rules</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ZedGraph</td>
</tr>
<tr>
<td>CheckParametersNullityInVisibleMethodsRule</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>239</td>
</tr>
<tr>
<td>Improved</td>
<td>366</td>
</tr>
<tr>
<td>AvoidUnusedPrivateFieldsRule</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>0</td>
</tr>
<tr>
<td>Improved</td>
<td>0</td>
</tr>
<tr>
<td>RemoveUnusedLocalVariablesRule</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>0</td>
</tr>
<tr>
<td>Improved</td>
<td>83</td>
</tr>
<tr>
<td>ArrayFieldsShouldNotBeReadOnlyRule</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>2</td>
</tr>
<tr>
<td>Improved</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rules</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ZedGraph</td>
</tr>
<tr>
<td>AvoidCaseSensitiveNamesRule</td>
<td>0</td>
</tr>
<tr>
<td>AvoidReturnSameValueRule</td>
<td>0</td>
</tr>
<tr>
<td>AvoidInfiniteLoopRule</td>
<td>0</td>
</tr>
<tr>
<td>AvoidUsingNullAfterNullityCheckRule</td>
<td>0</td>
</tr>
<tr>
<td>Total:</td>
<td>0</td>
</tr>
</tbody>
</table>
Extending Rule Set for Static Code Analysis in .NET Platform

7. Conclusion

Static code analysis is one of the cheapest and earliest ways to prevent or detect security flaws in software code. Therefore many commercial and open-source tools are developed for static code analysis. This paper investigates static code analysis tools for .NET platform. We have selected an open-source tool Gendarme which performs code analysis for compiled code and thus speeds up the analysis procedure compared with non-compiled code analysis. During compilation, the compiler itself removes a range of possible programming code errors or inconsistencies to standards. It is important to identify the errors that the compiler could not detect.

Static code analysis is typically based on a set of rules which define the nature of errors. In this paper we have proposed to extend Gendarme rule set by four new rules, which we consider important from our practical experience of software development. We have analyzed existing popular static analysis tools for .NET platform in order to determine whether they have the rules equivalent to our proposed new rules. The results show that none of the analyzed tools has the equivalents to all four new rules. These new rules should help detecting more excess, illogical code, which was not identified by the compiler.

We also investigated existing Gendarme rules and discovered that some of them could be improved. Therefore we have proposed improvements for four existing Gendarme rules. These improvements should help detecting errors and increasing program speed, stability, security, and code readability.

In order to assess the improvements made in Gendarme rule set in a real environment, the source code of five popular open-source programs was tested using new and improved rules. Results indicate that the improvements of existing Gendarme rules and the creation of new rules enabled detection of more errors and increasing quality of source code. 643 new errors were detected after analysis of 5 different applications. 429 of these errors were detected using improved rules and 214 errors were identified by the new rules.

Improvement of static code analysis is a continuous process. Therefore, in the future we are planning to analyze other possible code vulnerabilities and implement their detection by improving existing rules or creating new ones.

References


Received January 2015.