A Static Analysis Method for Detecting Buffer Overflow Vulnerabilities

Zhang Shuhao [*]
Nankai University
shuhao-zhang@outlook.com

Qin Jianxing [†]
Shanghai Jiao Tong University
qdelta@sjtu.edu.cn

Qu Shaobo [‡]
Huazhong University of Science and Technology
commcheck396@gmail.com

Hong Yun [§]
Xi'an Jiao Tong University
yoyoball@stu.xjtu.edu.cn

ABSTRACT

Buffer overflows are a weakness that is commonly found in some languages, such as C and Java. Any code without a strict boundary limit can pose a potential overflow risk and it's easy to be unaware of them. We demonstrate an approach to verification of C-like programs using analysis of the source code of programs. The approach applies a formal definition of the syntax and semantics of the subset of C and makes use of symbolic execution. Our approach is illustrated to capture all overflows of the source code written in our object language.

Keywords

buffer overflow, vulnerability in programs, static analysis, symbolic execution

1. INTRODUCTION

Buffer overflow is a typical vulnerability in programs. Many attacks on Microsoft systems are based on various buffer overflow problems. However, in some commonly used languages like C or Java, these problems will not be pointed out by the compiler.

In this paper, we will use symbolic execution to detect buffer overflow problems in small programs which are written in the subset of C and develop an analyzing application with a GUI.

[*]Zhang wrote the code of our core part and helped with the revision of our paper.

[†]Qin played the role as a leader. He provides most of the theoretical knowledge, wrote the code of our core part and helped with paper writing.

[‡]Qu wrote the code of our GUI and made our poster.

[§]I wrote most of this paper and made our video.

2. BACKGROUND

2.1 A glance at Buffer overflow attack

As its name suggests, buffer overflow occurs when a program attempts to write more data to a fixed-length block of memory or buffer than the buffer is allocated to hold[2]. Once an overflow occurs, the extra data will overwrite the data value in adjacent memory addresses of the destination buffer. This consequence can be used to crash a process or modify its internal variables. The attack that is based on this consequence is called buffer overflow.

This kind of attack can be fatal because the original data in the buffer includes the return pointer of the exploited function. The attacker can use the extra data to modify these pointers and modify the address to which the process should go next according to their own wishes. For example, they can make it point to a dangerous attack program and easily complete their attack.

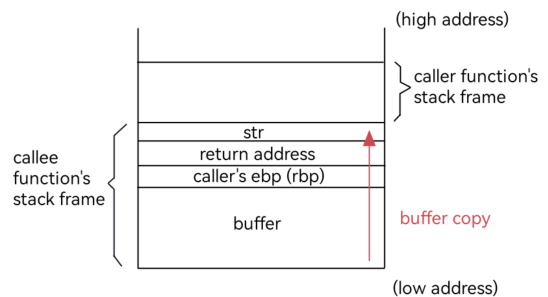

Figure 1: buffer overflow occurs when copying string str

Furthermore, buffer overflow attack ranks high in Common Weakness Enumeration (CWE)[2]. This sort of vulnerability can occur easily in programs without sufficient bounds checking. The preventing work can be sophisticated, especially when the occurrence of this kind of vulnerable problem will not always be warned by the compiler. As a result, an assistant tool with a function of checking potential overflow risk can be significant.

2.2 Symbolic execution overview

Symbolic execution[1] is a way of executing a program abstractly. This type of execution treats the inputs of the program symbolically and focuses on execution paths through the code. To be more specific, the interpreter will assume symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would[4], and the result of a symbolic execution should be expressed in terms of symbolic constants that represent the input values.

By doing an abstract execution, multiple possible inputs of the program that share a particular path can be covered[1]. In addition, symbolic execution also has another strength, that is, it can avoid reporting false warnings because each of all these errors represents a particular path through the program. This provides us an idea of checking potential buffer overflows. For any program statement that has the possibility of occurring buffer overflow problems, we can check the value range of those involved variables to verify whether they are satisfied the boundary constraints. If any of these variables fail to meet the condition, we can say that a buffer overflow problem occurs.

3. DEFINITION OF LANGUAGE

To simplify the process, we set the language of the source code to a subset of C. For the convenience of illustrating our ideas, we will use a less complicated language (we may call the language SymC--), which is semantically a subset of C, to describe our object language. Here are the detailed definitions.

Expression:

$$
\begin{array}{rll}
e ::= & n & \text{integer literal} \\
\mid & x & \text{name} \\
\mid & e \; bop \; e & \text{binary operation} \\
\mid & uop \; e & \text{unary operation} \\
\mid & \texttt{alloc}(e) & \text{memory allocation}
\end{array}
$$

Statement:

$$
\begin{array}{rll}
s ::= & \texttt{skip} & \text{empty statement} \\
\mid & \texttt{declint}\; x & \text{integer declaration} \\
\mid & \texttt{declptr}\; x & \text{pointer declaration} \\
\mid & x \leftarrow e & \text{assignment} \\
\mid & \texttt{touch}\; e[e] & \text{memory access} \\
\mid & s; s & \text{sequence} \\
\mid & \texttt{if}\; e \;\texttt{then}\; s \;\texttt{else}\; s & \text{conditional}
\end{array}
$$

Values:

$$
\begin{array}{rll}
v ::= & \texttt{int}\; v_i & \text{integer} \\
\mid & \texttt{ptr}\; v_i \; v_i & \text{pointer with bounds} \\
v_i ::= & n & \text{concrete integer} \\
\mid & x & \text{symbolic integer} \\
\mid & uop \; v_i & \text{unary operation} \\
\mid & v_i \; bop \; v_i & \text{binary operation} \\
\mid & \texttt{ite}\; a \; v_i \; v_i & \text{conditional}
\end{array}
$$

Assertions:

$$
\begin{array}{rll}
a ::= & v_i \; relop \; v_i & \text{integer relation} \\
\mid & \neg a & \text{negation} \\
\mid & a \wedge a & \text{conjunction} \\
\mid & a \vee a & \text{disjunction}
\end{array}
$$

It is also necessary to emphasize that this demo language sharing the same semantics with our source code language won't be involved in our code implementation. The purpose of developing such language is just for the convenience of illustration.

4. PROCESS OF ANALYSIS

The general process of analysis involves several parts.

The first part includes two preparation tasks. One is to check whether the source code can be compiled successfully. Another is to do an initial analysis with a parser and gain an abstract syntax tree of the code. The purpose of this step is to transform the code into an abstract form for the subsequent analysis.

In the second part, we will use symbolic execution to extract useful information from the result of the first step, and generate counterexamples of the variables involved in statements which have the risk of occurring buffer overflow.

Next, a SMT-solver will be used to check whether any of these counter examples can be satisfied. We will have a deeper discussion on how this solver can be used in part 4.3. With the feedback from the solver, we can locate the potential buffer overflows in the source code.

The last part of our analysis is to print the errors discovered to a GUI.

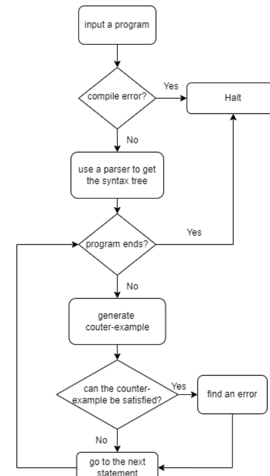

Figure 2: The whole process of the analysis[3]

Now, we will explain each part respectively.

4.1 Preparation

The first part of the preparation is to get the source code the user input and compile it. Our goal is to uncover those potential risks which cannot be warned by the compiler and our target source code should be those which can be compiled. If an error occurs, we will print the error information to the output box and halt the whole process.

Only if a source code can be compiled, will it be given to a parser to generate an abstract syntax tree. We use pycparser[1] here, which is a parser of C and written in Python. It can be replaced by any parser of C. The output will then be the input of the next part.

4.2 From program to counterexamples

The goal of this part is to generate counterexamples. In this part, we will use symbolic execution.

First, we will give a more specific illustration of this method.

When we do symbolic execution, we are actually executing a path through the source code. During this process, we will track the values of variables and the conditions that need to be satisfied for a specific path. It's necessary to mention that the values that we will keep track of are all expressed in terms of symbolic values. For example, if we have a user input variable a and execute $b = a + 1$, then the values we track should be

$$a \mapsto \alpha, b \mapsto \alpha + 1$$

In this example, α is a symbolic value, and $a \mapsto \alpha, b \mapsto \alpha + 1$ is what we will keep track of. We can use symbolic environment E as a collective name for the mapping from name to value that we keep track of.

To get a clearer picture of the process, we can take the code below as an example.

```
void foo(int *arr, int n, int head) {
    if (head > 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

In this function, we will track the value of arr, n, and $head$. Their symbolic constants are $ptr\ 0\ h', n'$ and $head'$. The symbolic environment E looks like this.

$$E = \{arr \mapsto \texttt{ptr}\ 0\ h',\ n \mapsto n',\ head \mapsto head'\}$$

When we meet conditional statement, we will track the condition for each branch. When we come to the first branch, the path condition P we will be tracking is $head' > 0$. For the second branch, the path condition P is $head' \leq 0$. The symbolic environment E is not updated in this function.

If there are any assumptions in the source code, it should be described by symbolic environment E and path condition P.

[1]You can find more information about pycparser at https://github.com/eliben/pycparser

```
void foo(int *arr, int n, int head) {
    ASSUME(n > 0, capacity(arr) >= n);
    if (head != 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

For example, if we add an assumption at the beginning of the function, the initial E and P should be look like this when after executing the assumption statement.

$$E = \{arr \mapsto ptr\ 0\ h',\ n \mapsto n',\ head \mapsto head'\}$$
$$P = \{n' > 0,\ h' \geq n'\}$$

Now we can introduce our idea of generating counterexamples with symbolic execution. For each memory access, the index should be always bounded in the range of the accessed array. This is our bounded requirement Q. Also, we have a path condition P which should be true when the program executes the corresponding statement. So it has to be guaranteed that $P \to Q$ in all cases, which implies $P \wedge \neg Q$ should not be satisfied. So we call $P \wedge \neg Q$ a counterexample.

Evaluation rules in the form of $E; e \Downarrow v$. It means expression e is evaluated to value v in environment E.

$$\frac{}{E; n \Downarrow \texttt{int}\ n} \qquad \frac{}{E; x \Downarrow E(x)}$$

$$\frac{E; e \Downarrow \texttt{int}\ v}{E; uop\ e \Downarrow \texttt{int}\ (uop\ v)} \qquad \frac{E; e_1 \Downarrow \texttt{int}\ v_1 \quad E; e_2 \Downarrow \texttt{int}\ v_2}{E; e_1\ bop\ e_2 \Downarrow \texttt{int}\ (v_1\ bop\ v_2)}$$

$$\frac{E; e_1 \Downarrow \texttt{ptr}\ l\ h \quad E; e_2 \Downarrow \texttt{int}\ v}{E; e_1 + e_2 \Downarrow \texttt{ptr}\ (l - v)\ (h - v)} \qquad \frac{E; e_1 \Downarrow \texttt{int}\ v \quad E; e_2 \Downarrow \texttt{ptr}\ l\ h}{E; e_1 + e_2 \Downarrow \texttt{ptr}\ (l - v)\ (h - v)}$$

$$\frac{E; e_1 \Downarrow \texttt{ptr}\ l\ h \quad E; e_2 \Downarrow \texttt{int}\ v}{E; e_1 - e_2 \Downarrow \texttt{ptr}\ (l + v)\ (h + v)}$$

$$\frac{E; e \Downarrow \texttt{int}\ v}{E; \texttt{alloc}(e) \Downarrow \texttt{ptr}\ 0\ v}$$

Execution rules in the form of $E, P; s \Downarrow E'; C$. It means given environment E and path condition P, after executing the statement s the environment will be updated to E' and counter examples C will be generated.

Path condition P is an assertion. C is the set of generated assertions that should not be satisfied.

$$\frac{}{E; n \Downarrow \texttt{int}\ n} \qquad \frac{}{E; x \Downarrow E(x)}$$

$$\frac{}{E, P; \texttt{declint}\ x \Downarrow E \cup \{x \mapsto \texttt{int}\ x'\}; \{\}}$$

$$\frac{}{E, P; \texttt{declptr}\ x \Downarrow E \cup \{x \mapsto \texttt{ptr}\ 0\ 0\}; \{\}}$$

$$\frac{E; e \Downarrow v}{E, P; x \leftarrow e \Downarrow E[x \mapsto v]; \{\}}$$

$$\frac{E, e_1 \Downarrow \texttt{ptr}\ l\ h \quad E, e_2 \Downarrow \texttt{int}\ v}{E, P; \texttt{touch}\ e_1[e_2] \Downarrow E; \{P \wedge (l > v \vee v \geq h)\}}$$

$$\frac{E, P; s_1 \Downarrow E'; C_1 \quad E', P; s_2 \Downarrow E''; C_2}{E, P; s_1; s_2 \Downarrow E''; C_1 \cup C_2}$$

$$\frac{E; e \Downarrow \texttt{int}\ v \quad E, P \wedge v \neq 0; s_1 \Downarrow E_1; C_1 \quad E, P \wedge v = 0; s_2 \Downarrow E_2; C_2}{E, P; \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \text{merge}(v, E_1, E_2); C_1 \cup C_2}$$

$\text{merge}(v, E_1, E_2) = E$ means $\forall x \in E_1, E_2$:

$$E_1(x) = \texttt{int}\ v_1, E_2(x) = \texttt{int}\ v_2$$
$$\rightarrow E(x) = \texttt{int}\ (\texttt{ite}\ (v = 0)\ v_2\ v_1)$$

$$E_1(x) = \texttt{ptr}\ l_1\ h_1, E_2(x) = \texttt{ptr}\ l_2\ h_2$$
$$\rightarrow E(x) = \texttt{ptr}\ (\texttt{ite}\ (v = 0)\ l_2\ l_1)\ (\texttt{ite}\ (v = 0)\ h_2\ h_1)$$

else error.

To get an intuition for what we can get from the process, we can take the code above (function $foo()$ with assumption) as an example.

After execution, the generated counterexamples will be

$$n' > 0 \wedge h' \geq n' \wedge head' \neq 0 \wedge (0 > 0 \vee 0 \geq h)$$
$$n' > 0 \wedge h' \geq n' \wedge head' = 0 \wedge (0 > n' \vee n' \geq h)$$

With the rules, we can write a program to realize this function.

4.3 Counterexample analysis

The goal for this step is to check whether any of the counterexamples we get from 4.2 can be satisfied.

Suppose that we now get a set of counterexamples C_i, $i = 1...n$. We should check each of them. If a particular counterexample can be satisfied, the line where we generate it could have overflow risk. Now the question has transformed to whether there exists a mapping from symbolic values to concrete values, such that $C_i = true$.

So far, we've transformed our original problem into a satisfiability problem. We can use SMT-solver z3[2] to solve it and find out the counterexamples which can be satisfied. The statement where we generate them are the errors we found.

5. GUI AND RUNNING RESULTS

After previous work, we have located the errors. The last stage of the analysis is to print these errors. In this part, we will provide an example code and its running results.

Our GUI is split into two parts. Users can type their source code waiting for analyzing to the left box and get the result from the box on the right.

Target source code

```
void foo(int n) {
    int *arr;
    if (n > 0) {
        arr = alloc(n);
    }
    print(arr[0]);
}

void get(int *arr, int n, int head) {
    ASSUME(n > 0, capacity(arr) >= n);
    if (head != 0) {
        print(arr[0]);
    } else {
        print(arr[n]);
    }
}
```

[2]You can find more information about z3 at https://github.com/Z3Prover/z3/wiki#background

The running result is like this.

Figure 3: Running result of the example code above.

Our analyser returns two errors. They're highlighted and provided with counter examples that can be satisfied.

6. RELATED WORKS AND COMPARISON

In the method we mentioned above, we're actually doing static analysis. There's also another way to test vulnerability of programs, which is called dynamic analysis. Unlike static analysis, dynamic analysis evaluates the program by executing data in real-time. The input cases can be generated by a program using random algorithm, or be typed in by the tester.

What these two kinds of methods have in common is that they're both trying to find counterexamples. Though dynamic analysis can prove a program has overflows, it can be overwhelmed when faced with a correct program. The method we mentioned, on the other hand, uses static analysis and attempts to find counterexamples in a more theoretical way, which allows it to prove the non-existence of counterexamples.

Besides, our method has another advantage over dynamic analysis in efficiency. While a particular input case can only find out some of the buffer overflows, symbolic execution can check every possible path in the program. This allows us to find all the errors with only one execution instead of testing endlessly.

7. LIMITATIONS

Our project is a very basic prototype. It has several known limitations shown in our semantics model.

- The values in the array are discarded.
- Loops are not supported in the language.
- The program has only one exit point at the end.

8. CONCLUSION

From the represented results we can see that our method has advantages of precision, efficiency and automation. However, there are also some shortcomings. First is that the solver we use in counterexamples analysis (4.3) is unstable. The reason is that the satisfiability problem we are trying to solve is a NP-hard problem. The second shortage is that since the process gives another program as the input of our program, it can't handle any program according to the halting problem.

However, in most of the cases, our method can perform well and give a correct result quickly. So we can still say that the advantages outweigh the disadvantages and this method can be of help in testing programs for vulnerabilities.

9. ACKNOWLEDGEMENT

I would like to express my very great appreciation to Professor Hugh Anderson and his teaching assistant Harish Venkatesan for their valuable and constructive suggestions during the planning and development of our project.

I would also like to thank my group-mates. They did the really hard work of our coding part and gave me useful suggestions in paper writing.

References

[1] J Aldrich. *Symbolic Execution (Program Analysis lecture notes - Spring 2019)*. https://www.cs.cmu.edu/~aldrich/courses/17-355-19sp/notes/notes14-symbolic-execution.pdf. 2019.

[2] Michael Cobb. *buffer overflow*. https://www.techtarget.com/searchsecurity/definition/buffer-overflow.

[3] Wenliang Du. *Computer Security:A Hands-on Approach*. Wenliang Du, 2019.

[4] Wikipedia. *buffer overflow*. https://en.wikipedia.org/wiki/Symbolic_execution.