# Staticly Detect Stack Overflow Vulnerabilities with Taint Analysis

Zhang XING[1] , Zhang BIN[1] ,Feng CHAO[1] and Zhang QUAN[1]

[1]*School of Electronic Science and Engineering , National University of Defense Technology, ChangSha, China*

**Abstract.** Nowadays binary static analysis uses dangerous system library function to detect stack overflow vulnerary in program and there is no effective way to dig out the function which can cause stack overflow issue. List necessarily characteristics of the function which may cause stack overflow vulnerary and define stack overflow dangerous function(SODF). Then introduce static taint analysis to detect SODF include taint introduction, taint propagation and taint checking stragety. Next describe the particular process of detecting SODF in the program with static taint analysis. Finally choose 4 runtime library and 2 binary software, and detect whether the chosen software has SODF and locate the name of SODF with static taint analysis. Testing result shows that the algorithm can detect and locate plenty of SODF in test program which means the algorithm can work efficiently.

## 1 Introduction

Stack overflow is a well-known security issue, which can be used to execute unauthorized instructions and illegal operations. The study of stack overflow vulnerability detection is of great significance.

Current method of detecting stack overflow vulnerabilities can be divided into dynamic and static analysis. Dynamic analysis, mainly through checking data-integrity of an array of borders and return pointer, whose typical tools include StackGuard [1] and RAD [2]. Overall, stack overflow is often detected when the stack has been crashed.

Source code analysis and binary analysis provide effective performance in static analysis. Source code analysis detects possible vulnerabilities by comparison with known vulnerabilities after lexical analysis. Wherein, Flawfinder[3] and RATS[4] are typical analysis tools. Binary analysis aims at checking if there is boundary detection when calling library functions to determine whether exists stack overflow vulnerability.

Static analysis can efficiently locate the stack overflow vulnerability. And dynamic analysis can be more effective with the prior knowledge provided by the static analysis. However, it is very difficult to access most of the program's source code. And there is no good way to locate the function which has stack overflow risk and defined by the program itself.

Most static method to detect binary software's stack overflow vulnerability is based on analyzing known vulnerable library functions. However many programs have stack overflow vulnerability because of bad boundary condition of self-defined function. IKEView.exe has stack overflow vulnerability because of a self-define function with R60 version.

In this paper, we establish a model for stack overflow dangerous function, and propose an algorithm to detect stack overflow vulnerabilities. We have implemented our method and applied it to test several binary software. The results show that static taint analysis can achieve higher accuracy.

## 2 Basic Definitions

Despite the formats of executable files on different operation system are different, but the basic formulation methods are consistent.

Executable file contains import and export tables, global variables and functions. Function consists of basic blocks, which are made up of basic instructions. Here we give out some basic definitions in binary software.

### 2.1 Some Definitions

**Instruction**: Instruction is instructing a computer to perform a certain operation command, abbreviate as $I$ , and it is the smallest unit of computer-implemented. It consists of a string of binary number, usually including opcode and operands. Opcodes represents the operation which the instruction is going to operate. Operands represent particular data which is operated by the instruction.

**Basic Block**: Basic Block is an execution sequence of instructions, abbreviate as *B* ,which has only one entrance and one exit. The entrance is the first instruction of the basic block while the exit is the last. Therefore, as long as the first instruction is executed in time, then all other instructions within a basic block will be executed once the order.

**Basic Block** Set: Basic Block Set is composed of several basic blocks that can access to each other, abbreviate as *BB* .

**Function**: Function is a basic unit of an executable module, and a combination of basic blocks that accomplish a specific function together, abbreviate as F.

### 2.2 Intermediate Languages

It is very difficult to analysis disassembled assembly code directly due to the wide range of assembly instructions. Normally we lift the disassembled assembly code into intermediate language.

Intermediate language abstracts form native assembly code and therefore makes it easier to develop analysis tools and algorithms.

In this paper, we use the REIL(Reverse Engineering Intermediate Language), a platform-independent intermediate language to represent disassembled assembly code.

## 3 Stack Overflow Dangerous Function Model

So far, most static code analysis methods decide whether a program has stack overflow vulnerability by looking for known dangerous functions such as *strcpy* and *memcpy*. However, the source of stack overflow is not the function name itself, but their approach to the data. For example, the dangerous function copies the source data into the destination buffer without correctly checking the size of the destination buffer.

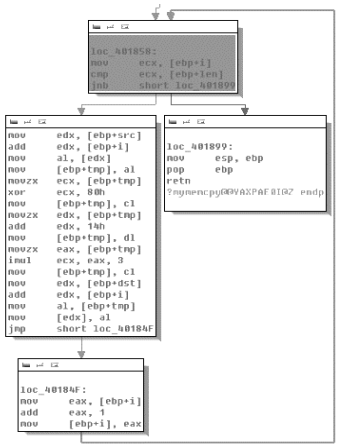


**Figure 1** The Disassembly Code of the Function

Parameter *src* is the address of source buffer and parameter *dst* is the address of destination buffer of the function. And data transfer from the source buffer to the destination buffer through local variable *tmp* in the loop. The loop control variable *i* is independent on the destination buffer. So *mymemcpy* function is a SODF. The function is an unknown SODF, and many static analysis methods often ignore it.

### 3.1. Characteristic of SODF

Always, data transfer between source buffer and destination buffer in a loop, and the control condition of the loop depends on the source buffer. There are many self-defined functions that have the above characteristics, and they are likely to become internal factors for stack overflow.

Through the analysis of a large number of stack overflow vulnerabilities, we summarize the characteristics of Stack Overflow Dangerous Function as follow:

(1) Contains a loop

(2) Data transfer from the source buffer to the destination buffer in the loop

(3) The source buffer address is the parameter of the function

(4) The destination buffer is the parameter or a local variable of the function

(5) The loop control variable is not dependent on the destination buffer..

Next, we propose our static taint analysis method to locate that kind of function.

## 4 Detecting Stack Overflow Dangerous Function with Static Taint Analysis

### 4.1. Over View

In this section, we leverage static taint analysis to locate stack overflow dangerous function (SODF). And introduce

an algorithm based on static taint analysis to analyze program.

The algorithm is divided into two phases: cyclic basic block set's filter and static taint analysis. The cyclic basic block set's filter (step one to step three) aims at locating the possible dangerous function address. Static taint analysis (step four to step five) checks whether data transfers from buffer to buffer in the loop and determines whether the function is a dangerous one.

General idea of the algorithm is as follows:

Step one, get the function fwith_args which introduces the parameter in the program.

Step two, locate loop basic block set BBwith_cycle which contains cycle relationship in fwith_args.

Step three, remove the jump instruction Jcc in every BBwith_cycle and form new basic block Bneed_analysis.

Step four, introduce taint for each basic block Bneed_analysis. If the basic block invokes the parameter of the function, then we mark the parameter as taint and on to the next step.

Step five, we translate the $B_{need\_analysis}$ into intermediate language basic block and start the taint propagation. If the destination buffer is tainted, then we can say the function is a stack overflow dangerous function.

## 4.2. Suspicious Basic Block

Suspicious basic block is a basic block set. It is a set of all possible basic blocks between loop start and loop end in the loop basic block set. We set the begin of a loop basic block set is the lowest address of the set and the end of a loop basic block set is the highest address of the set.

We adopt a conservative analysis strategy in static taint analysis. So we analysis all possible paths from loop starts to loop ends, and can effectively improve the accuracy of the algorithm through this method.

The mymemcpy function corresponding to that shown in Figure 1, its Suspicious BB is {0x401858, 0x401860, 0x40184F}.

## 4.3. Static Taint Analysis

### *4.3.1 Static Taint State*

Static taint analysis does not real execute target program, there is no runtime information. We propose static taint state to introduce taint and taint propagation.

Every operand has two static taint state, the operand state $T:t\to Bool$ and memory state $M:t\to Bool$. State $T$ and state $M$ each has two states : tainted state and untainted state.

Before the propagation starts, all operands' static taint state are set to untainted.

### *4.3.2 Taint Analysis Strategy*

Taint analysis strategy includes taint introduction, taint propagation and taint checking. It mainly checks whether to-be-analysis-basic-block-set transfers parameter of the function as the source buffer to destination buffer.

(1) Taint Introduction

Taint introduction here mainly concerns about the parameters of the function. According to the stack structure, parameters of the function get indexed by EBP/RBP register. When instruction introduce the parameter in the memory, then we mark the operand of the instruction as taint source, and set the operand state and memory state tainted.

(2) Taint Propagation

$T[t]$ indicates that intermediate instruction's operand $t$ 's taint state. $M[t]$ indicates that intermediate the memory's taint state pointed to instruction's operand $t$. $a\mapsto$ indicates that operand $a$ 's taint state updates for operand $b$ 's taint state. $TM[a\mapsto$ indicates that operand $a$ 's operand and memory state updates for operand $b$ 's operand and memory state.

To express taint propagation process, Table 1 shows the intermediate instructions' operational semantics, where $P_{xxx}$ represents the taint propagation strategy. And taint propagation strategy is shown in Table 2.

**Table 1** Taint Operation Semantics

| Instruction | Taint operation semantics |
|---|---|
| *ADD t1,t2,y* | $TM[y\mapsto$ $,T[t_2])]$ |
| $SUB\,t_1,t_2,y$ | $TM[y\mapsto$ $t_1],T[t_2])]$ |
| $MUL\,t_1,t_2,y$ | $TM[y\mapsto$ $,T[t_2])]$ |
| $DIV\,t_1,t_2,y$ | $TM[y\mapsto$ $,T[t_2])]$ |
| $MOD\,t_1,t_2,y$ | $TM[y\mapsto$ $,T[t_2])]$ |
| $SHL\,t_1,t_2,y$ | $TM[y\mapsto$ $,T[t_2])]$ |
| $AND\,t_1,t_2,y$ | $TM[y\mapsto$ $,T[t_2])]$ |
| $OR\,t_1,t_2,y$ | $TM[y\mapsto$ $,T[t_2])]$ |
| $XOR\,t_1,t_2,y$ | $TM[y\mapsto$ $t_1],T[t_2])]$ |
| $LDM\,t_1,e,y$ | $T[y\mapsto$ $[t_1])]$ |
| $STR\,t_1,e,y$ | $TM[y\mapsto$ $t_1])]$ |
| $BISZ\,t_1,e,y$ | Undefined |
| $JCC\,t_1,e,y$ | Undefined |
| $UNDEF\,e,e,e$ | Undefined |
| $NOP\,e,e,e$ | Undefined |
| $UNKN\,e,e,e$ | Undefined |

**Table 2** Taint Propagation Strategy

| Propagation Strategy | Detail |
|---|---|
| $P_{or}$*(T[t1],T[t2])* | Returns tainted state if $t_1$ or $t_2$ is tainted. |

| | |
|---|---|
| $P_{xor}(T[t1],T[t2])$ | Returns untainted state if $t_1$ and $t_2$ are the same, otherwise strategy is same as $P_{or}(T[t_1],T[t_2])$. |
| $P_{m2r}(M[t1])$ | Returns $t_1$'s memory taint state, and the destination operand must be operand taint state. |
| $P_{r2r}(T[t1])$ | Returns $t_1$'s operand taint state, and the destination operand must be operand taint state. |

(3) Taint Checking Strategy

When instruction *STM* is executed, taint checking begins. Algorithm checks source operand's operand state and memory state. The function is stack overflow dangerous function if the operand's memory state is tainted.

The loop basic block set is extracted from the program in section III. Then it transformed to *Suspicious BB*. And the *Suspicious BB* is translated to the REIL basic block. The specific static taint analysis progress with the REIL basic block is shown below.

We can know from the Code II that the REIL basic block read data from memory pointed by parameter of function, and transfer the data to destination buffer. Thus, *mymemory* is an SODF.

The algorithm adopts conservative strategy, so it will costs a lot of time to analyze functions which has high cyclomatic complexity. State explosion problem will happen when cyclomatic complexity is higher than 50. We ignore those functions when practical analyzing program.

# 5 Testing

## 5.1. Experiment Design

This chapter chooses 4 runtime libraries and 2 software, which is shown in **Table 3**. And the detection of SODF is a new technique nowadays ,so no comparison with other techniques.

Static taint analysis algorithm is based on plugin of IDA pro. The plugin runs on Windows 7 SP1 x32 system, processor is i5 2510m@2.5GHz, and memory is 3GB.

**Table 3** Specific Information of Library and Software

| Name | Version | description |
|---|---|---|
| npFoxitReaderPlugin.dll | 2.2.5.107 | A dll of foxit reader,it is used to load plugin |
| Wpsio.dll | 9.1.0.5041 | A dll of WPS,it is used to process vsd/vsdx file |
| ffmpeg.dll | 3.11.4.16 | A dll which is used to handling multimedia data, and it is widely used |
| msvcrt.dll | 16385 | The Microsoft Visual C Run-Time Library for Visual C++ version 4.2 to 6.0 |
| VulnApp.exe | - | A program which contains a self-defined stack overflow dangerous function |
| IKEView.exe | R60 | A program which can diagnosis VPN's fault |

## 5.2. Result

The result of the algorithm is shown in **Table 4**. Nk represents the known number of stack overflow dangerous function[NVD, CVE]. Nr represents the real number of stack overflow dangerous function. Ni represents the number of functions that algorithm ignore to analysis due to high cyclomatic complexity. Nc represents the number of repetitions that known SODF and detected SODF. Time represents the time that the algorithm performs one time costs.

**Table 4** Analysis Result

| Target Program | Nk | Nr | Ni | Nc | Time (min) |
|---|---|---|---|---|---|
| npFoxitReaderPlugin.dll | 0 | 18 | 14 | 0 | 26 |
| Wpsio.dll | 0 | 23 | 5 | 0 | 35 |
| ffmpeg.dll | 0 | 20 | 71 | 0 | 87 |
| msvcrt.dll | 14 | 24 | 56 | 14 | 125 |
| VulnApp | 1 | 2 | 0 | 1 | 2 |
| IKEView.exe | 1 | 2 | 1 | 1 | 13 |

**Figure 2** shows that the SODF that the algorithm detected. We can see that data stored in the address that pointed by arg_0 transfers to the address pointed by arg_4 in the cycle structure. And loop variable arg_8 is independent with destination buffer. So we can say the function is a SODF. And in practical the function is the cause of stack overflow vulnerability in IKEView.exe.

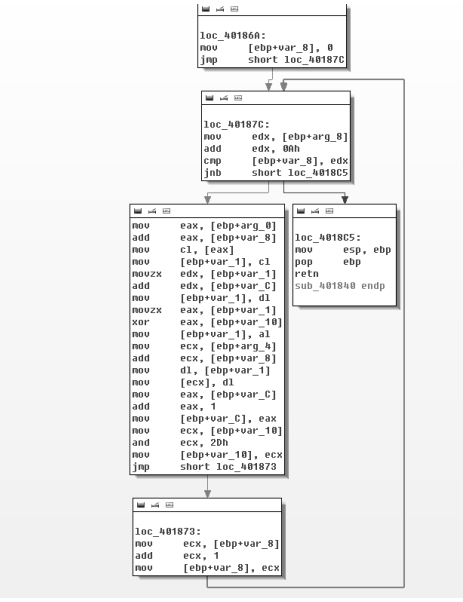


**Figure 3** Disassembly Code in Ikeview.Exe

It will take quite long time to analysis the function that has high cyclomatic complexity, so the algorithm ignore those functions. However, the static taint analysis algorithm can accurately locate the SODF as shown in the result.

## 6. Conclusions

We proposed static taint analysis algorithm based on stack overflow vulnerabilities' behavior. Experimental result shows that algorithm can effectively locate self-defined SODF in program. However, algorithm will cost lots of time when analyzing functions with high cyclomatic complexity. Next, we will continue optimize the efficiency of the algorithm. Further static taint analysis algorithm can be used on other vulnerabilities' pattern.
The algorithm can be combined with dynamic analysis. By providing a possible position of SODF, making dynamic analysis more effectively to monitoring program.

## References

1. Cowan C, Pu C, Maier D, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks[C]//Usenix Security. 1998, 98: 63-78.
2. Tzi-Cker Chiueh,Fu-Hau Hsu.RAD:a complie-time solution to buffer overflow attacks.Distributed Computing Systems,2001.21st International Conference on 08-12Jan.2006,pages 1-7
3. Ferschke O, Gurevych I, Rittberger M. FlawFinder: A Modular System for Predicting Quality Flaws in Wikipedia[C]//CLEF (Online Working Notes/Labs/Workshop). 2012.pages 1-10.
4. RATS. http://www.forifysoftware.com/security-resources/rats.jsp.
5. Lattner C,et al. The LLVM compiler infrastructure [J].The LLVM compiler infrastructure 2010.pages 1-8.
6. Nethercote N,Seward J.Valgrind: a framework for heavyweight dynamic binary instrumentation [C].In ACM Sigplan Notices.2007:89-100.
7. Song D,Brumley D,Yin H,et al. BitBlaze:A new approach to computer security via binary analysis[M].Information systems security.Spring,2008:2008:1-25.pages 2-5.
8. Brumley D,Jager I,Avgerinos T,et al.BAP:A binary analysis platform[C].In Computer Aided Verification.2011:463-469
9. Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004: 75-86.
10. Dullien T, Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis[J]. Proceeding of CanSecWest, 2009.