Integer Overflow Vulnerabilities Detection in Software Binary Code

Roman Demidov
Peter the Great St. Petersburg Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
rd@ibks.spbstu.ru

Alexander Pechenkin
Peter the Great St. Petersburg Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
alexander.pechenkin@ibks.ftk.spbstu.ru

Peter Zegzhda
Peter the Great St. Petersburg Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
zeg@ibks.ftk.spbstu.ru

ABSTRACT

In this paper[1] we propose a new approach to detect integer overflow vulnerabilities in executable x86-architecture code. The approach is based on symbolic execution of the code and the dual representation of memory. We build truncated control flow graph, based on the machine code. Layers in that graph are checked for the feasibility of vulnerability conditions. The proposed methods were implemented and experimentally tested on executable code.

CCS CONCEPTS

Security and privacy → System security → Vulnerability management

KEYWORDS

Vulnerability finding, symbolic execution, symbolic memory, vulnerability classification, control flow graph, integer overflow

1 INTRODUCTION

Vulnerability detection is complex computational problem. In general this problem can be reduced to NP-complete problem Boolean satisfiability problem (SAT). The exact solution of this problem does not exist. All actual algorithms are based on exponential enumeration of possibilities. As a result the implementation of this approach cannot be effective in practice. There are variety of vulnerability detection technics which do not involve complete enumeration. Heuristic approaches focus on a-priori known vulnerability characteristics detection rather than vulnerability detection itself. Consequently, these methods have a narrow scope of applications defined by a class of the input programs. As a result universal approach which allows automatically detecting vulnerabilities of known classes in actual programs does not exist.

Practical methods have number of disadvantages. Fuzzing might be considered as a dynamic analysis method. The various options of fuzzing use prior knowledge of input data format and algorithm implementation bottlenecks. This allows reducing number of scope for complete enumeration [1]. Effective fuzzing requires substantial computational resources [2, 3] without guaranteed results. Fuzzing does not make possible to find all vulnerabilities in the program code. Furthermore this method is inappropriate for specific class vulnerability detection.

Static analysis methods make use of symbolic execution. A part of the memory cells is supposed to be symbolic i.e. unknown. A set of constraints on symbolic values is built for potential vulnerability paths. Satisfiable set of constraints shows that vulnerability exists. For the first time idea of symbolic execution appears in the eighties of XX century [4, 5]. Application of basic method might cause numerous difficulties. A large set of constraints requires substantial computational resources. The number of interesting paths may be exponential. Another issue is an existence of cycles in control flow graph. Number of cycle traversal depends on symbolic values. Furthermore decision of complex set of constraints can be inappropriate for the required path. This problem is caused by incompleteness of third party effects consideration in set of constraints.

This article proposes partial solutions for problems occur in the search of vulnerabilities using symbolic code execution. We introduce some new approach, which can significantly help to find a range of additional vulnerabilities during testing process.

2 RELATED WORK

A wide range of research was previously made due to address issues of symbolic execution method for executable code. Researchers from BitBlaze project (Berkley University, USA)

SIN '17, October 13–15, 2017, Jaipur, IN, India

ACM ISBN 978-1-4503-5303-8/17/10..$15.00
https://doi.org/10.1145/3136825.3136872

propose to resolve the problem of cycles with symbolic number of steps [6]. Proposed method is based on cohesive variable usage. The authors of the article use number of completed iteration in a cycle as a cohesive variable. Index variables inside of cycle can be computed using this variable. That allows building set of constraints without complete enumeration of all possible iteration numbers. However this approach is suitable only for simple dependencies of variables in loop body. The authors of paper [7] proposed selective symbolic code execution for large software complexes. Later this idea was used in systems called Driller [8], Mayhem [9]. It seems to be the right way to combine static and dynamic analysis in one approach. But this approach inherits all issues of fuzzing – scalability problem, the hardness of test case properly generation, error-based only vulnerability detection. The other important example is Springfield project [3] which was developed by Microsoft research unit. Proposed method is based on concolic code execution and parallel computations in high-performance cloud. Programs under investigation execute with an actual input data but this data is generated using symbolic execution and constraint solving. Upon the application of Microsoft their implementation of automatically theorem proving system allow using set of constraints with billionth size symbolic variables successfully. But this project is based on a huge computational architecture, which is not available in ordinary companies for private closed-source testing. So he is also could not be fully applied for integer overflow vulnerabilities detection in executable x86-architecture code.

In comparison with others, our approach allows to detect memory corruptions at an early stage, in addition to the main task of integer overflows discovery. Our memory management mechanism simplifies that process by lazy memory initialization and runtime tracking the types of memory used.

3 SYMBOLIC EXECUTION APPLICATION FOR INTEGER OVERFLOW VULNERABILITY DETECTION

Proposed approach allows finding integer overflow vulnerabilities in binary code and other types of software errors such as uninitialized memory usage, numerical values dereferencing instead of address dereference, inappropriate operations with addresses, etc.

The main idea of approach is to use symbolic code execution for vulnerable conditions construction for input variables. Afterwards these condition need to be solved. Proposed approach can be presented as the sequence of following phases:

1. Control flow graph construction. The input file is used to construct control flow graph for the tested program. Additionally the input and output vertices should be detected. The input vertices represent parts of code which are the sources of external data (keyboard input, file reading, etc.). The output vertices are code parts where “dangerous functions” calls exist. “Dangerous functions” are memory allocation functions.
2. Cutting of unused path in control flow graph. All vertices which are not used in any path from input vertex to output vertex should be cut.
3. Symbolic entry construction. Input data and uninitialized memory cells are considered as symbolic. Every cell is represented as a pair including symbolic “number” and symbolic “address”.
4. Symbolic emulation. Every path is processed independently. The process includes machine code instructions symbolic emulation. This process affects both part of the cells representation (number and address). The symbolic cells may lose one of these parameters if any operation is impossible. For example, addresses could not be multiplied by any value and numbers could not be dereferenced. Associated with representation conflicts finding allows to detect vulnerabilities or software errors.
5. Condition system construction. The condition system describes approachability of output vertex from input vertex. The system is constrained during the paths exploration. Special condition is used as a last condition in the system. If this condition is met, the integer overflow occurs.
6. Condition system feasibility checking. Created conditions are checked with the automatically theorem proving system. If system is feasible program is vulnerable. Satisfying set of system form vulnerable input data called proof of concept which is a provement for this vulnerability.

Python language and additional software tools were used for experimental implementation of this approach. Implementation details are described in the following sections.

4 CONTROL FLOW GRAPH CONSTRUCTION

IDA Pro disassembler is meant to be used as a primary data source for the execution file. IDA allows getting disassembled code with the instructions identification. The built-in tool IDAPython makes it possible to construct control flow graph. The process involves separate blocks detection where every block is free of conditional branch operations. IDAPython is a powerful code analyses tool however there are substantial design constructions:

– Built-in Python interpreter is suitable only for 32-bit mode. This fact imposes restrictions on accessible for analyses amount of memory (4 GB).

– Plugin is unable to work outside the IDA Pro interactive console. This situation complicates possible parallel path processing in graph.

In that regard the PaiMei framework was improved. This framework is an IDAPython script which is used to construction and code graph representation extraction to the file.

Basic code blocks are used as the vertices of extracted graph. These basic blocks represent homogeneous code regions which are free of branch instructions, function calls and function return instructions (Figure 1).

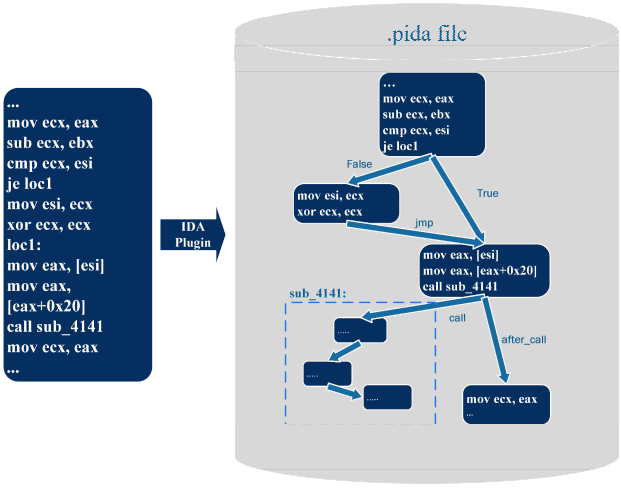

Figure 1: Retrieving information from IDA Pro in graph form

The branch conditions apply to form set of directed edges. Every edge is marked with a label which depends on last instruction of base block:

– *True/False*: label is used if last instruction was a branch condition next two blocks are marked with the branch condition corresponding label,

– *jmp:* this label is used in case of unconditional branch (not to the next instruction),

– *jmp_import:* in case of jump to import section function,

– *call:* is used if the last instruction was a function call,

– *after_call:* if last instruction was a function call, an edge with this label is added to current block (not called function block).

The labels will be used for symbolic variables condition construction during path proceeding.

Extracted file will contain all required for analysis information about code. Consequently the future work will be based on with this extracted file usage rather than executable file employment.

5 GRAPH RECTIFICATION

It is necessary to find graph path which leads from input data to vulnerable areas. Basic blocks with external data reading will be used as input vertices. Account is taken of either reading from files (fread, scanf, recv), from command prompt, from register, etc., or taking external parameters as function arguments/previously initialized values. For the standalone code (exe files etc.) with the one entry point the first variant is more suitable. However, the second variant is more suitable for dynamic libraries (dlls, dylib etc.), where many functions can be called independently from the programs which use the target libraries. Blocks with memory allocation (malloc, new, HeapAlloc, etc.) will be used as output vertices. The names of functions apply for vertex identification. The graph rectification process can be divided into two sections (Figure 2):

1. Recursive Depth-First-Search (DFS). Vertices which are included in interesting paths are marked colored. Algorithm will finish after the last start vertex recursion terminate. The algorithm terminates after the exit of the recursion in the last of the starting vertices.
2. All uncolored vertices and adjacent edges should be removed from the graph. Remaining colored paths in the graph could be processed independently from each other.

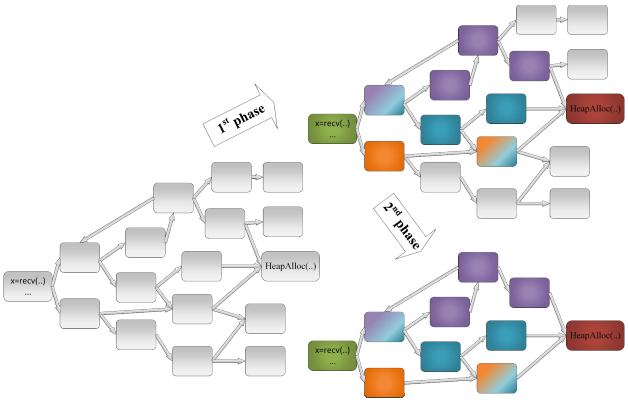

Figure 2: Control flow graph rectification with path detection

6 PROGRAM MEMORY INTERNAL REPRESENTATION

To analyze the program without its execution we propose to model uninitialized and external data as symbolic variables. The z3py framework applies as a base tool for symbolic values proceeding. This framework allows constructing complicated set of constraints and solving it effectively.

It is necessary to form consistent vulnerability appearance conditions with the purpose of consistent conditions solving. This requires memory operations emulation. The following facts should be appreciated:

– x86 architecture does not make a difference between numbers and addresses.

– There is no information about variable types. Value treatment as signed or unsigned is shown only implicitly (near conditional branch instructions).

– Uninitialized variables contain arbitrary values in high-level programming languages (C, C++). This applies to corresponding memory cells during compilation. This memory considered to be external data source in case of symbolic execution.

Proposed memory model appreciate this and other requirement. The memory is represented as an associative array. The lazy initialization is applied: memory cells will be created the first time when it will be use.

The list of possible memory cells:

– general purpose registers (“eax” - .. – “edi”);
– *local values* on current function stack ([ebp+var_8]);
– current function *arguments* ([ebp+arg_0]);
– values from allocated dynamic memory buffer;

– memory areas located near symbolic address.

Initially the analyzed path memory is empty. When memory management instruction is achieved, corresponding cell existent will be checked. If this cell does not exist it will be created with a new symbolic value production. If the cell is not used for reading this checking is not implemented.

The content of created cells has a *dual representation*. The way cell will be used (as number or as address) is not known in advance. Therefore the symbolic execution influence both part of the representation (number part and address part) (Figure 3). This has been going until first instruction which allows determining cell content type (cell may contain number or address) would be achieved. Cell content which is meant to be used with arithmetic or bit operations will be interpreted as a number values. In this way, the attempt to dereference this cell indicates vulnerability existence. Instead, cell which value is used as a base in an indirect memory dereferencing could not be used as a numerical value during the current path proceeding.

Figure 3 shows corresponding example. The ebx register was considered as a dual represented before dereferencing occurs. After dereference, register content is thought of address until current path is explored. This way detected *value* incorrect *usage* indicates possible vulnerabilities.

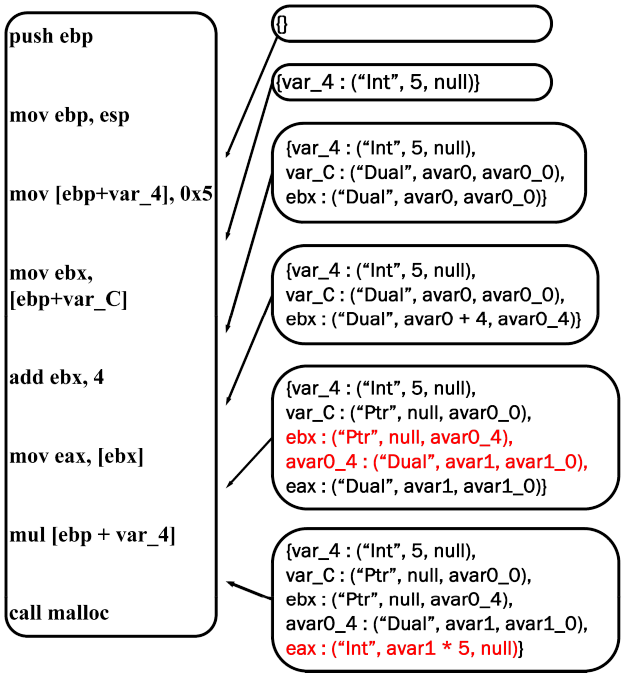

Figure 3: Duality of memory processing during symbolic emulation

7 SYMBOLIC EXECUTION

Symbolic code emulation and code affected memory modification apply to each leading to destination vertex path in graph. At the same time condition for symbolic values initiating integer overflow may accumulate in some code areas. This can be following areas:

– edges between vertices in the control flow graph, caused by conditional branches. Expression which representing conditional branch is added to corresponding variables condition system;

– arithmetic instructions: add, sub, mul/imul, shl, etc. Added expression presents conditions which cause target cell overflow in any operation involved in path to output vertex (Figure 4).

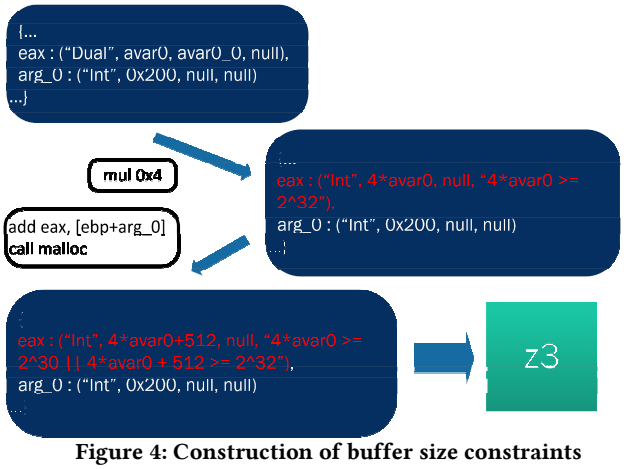

Figure 4: Construction of buffer size constraints

When the output vertex is reached, set of constraints is retrieved from corresponding memory cell. After that state-of-the art theorem prover Microsoft z3 (open source) is applied to check if solution for this set exists. Solution existence indicates that there is vulnerability. Satisfying set of construction system represents vulnerable set of external or uninitialized data which leads to integer overflow in current path.

Depending on the kind of analyzed code symbolic execution can take place in two different modes:

1. Full-paths mode. Accumulation of system conditions occurs in the forward direction when moving from the source to the destination node. In this case, system of vulnerable constraints imposed only on the external data of the program and on uninitialized variables when moving along each path.
2. Partial-paths mode, or layer-by-layer mode. This mode allows to analyze code, where there is no single entry point into the program – for example, DLL libraries. Various library functions can run independently from each other by different programs with different parameters. This implies the need for vulnerability testing of individual functions of such libraries.

   Function call stack is formed along a given path upon reaching the end vertex using the depth-first search. This allows to work with the functions in the reverse order of their execution along a given path. In the layer-by-layer mode symbolic execution of the code starts with the top (the innermost) function on the call stack. Upon reaching the destination vertex the generated system of vulnerable

conditions is checked for solvability. In the case of solvability, the function is considered as vulnerable and the symbolic execution performs for the following function on the call stack (Figure 5). Next destination node in this case will be the place a call to the vulnerable function.

As before, the conditions of the destination node reachability are accumulated. The set of constraints to verify the current function is complemented by a set of constraints that link the symbolic variables of the affected function with symbolic values for these parameters in the emulation round of the current function. In addition to the parameter values, all the symbolic values are recursively binding in the current and vulnerable functions, which are descendants of these values and are initialized through their dereference.

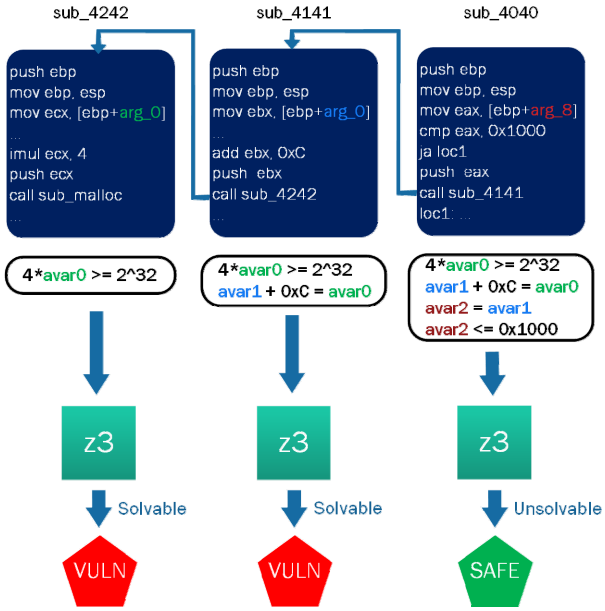

Figure 5: Layer-by-layer vulnerability checking

One can observe (Figure 6) an example of layer-by-layer mode operation for 2 functions. A set of vulnerable constraints, built for function sub_4040 contains 3 types of constraints:

- constraints for calling function sub_4141,
- constraints describing the reachability of a function call sub_4141 from the current function sub_4040,
- constraints, linking the symbolic values that are actually passed to a function sub_4141 from current function sub_4040 with the symbolic variables of these parameters.

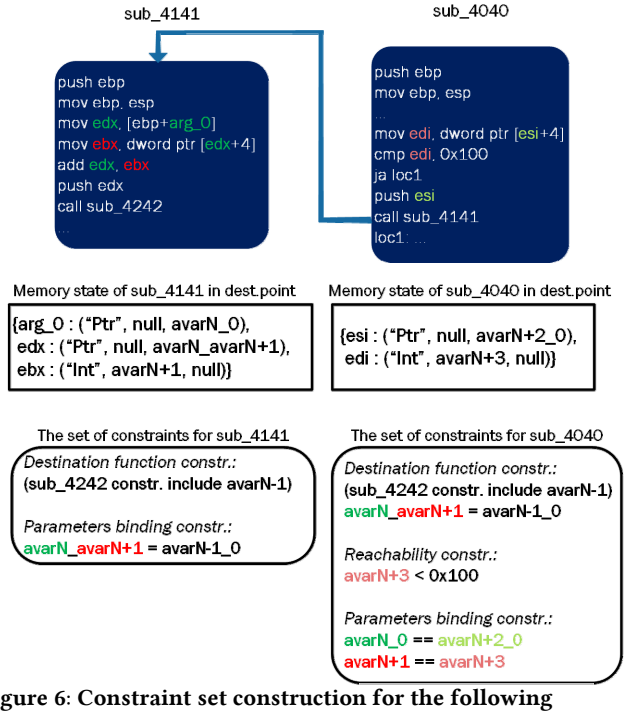

Figure 6: Constraint set construction for the following function from the call stack

System constraints are checked for solvability sequentially from the innermost function. If the system is solvable, it indicates that this function is vulnerable. If at any stage of the computation process, the constraints system become insolvable, a further test along this path is not performing.

8 CONCLUSION

This approach was implemented to optimize existing solutions and approbate new approaches in case of memory vulnerabilities detection. The method was tested on the program executable code which was borrowed from educational resource for the "Protocol security analysis" course in Saint-Petersburg State Polytechnic University. In simple cases method was able to detect integer overflows, additional vulnerabilities of uninitialized variables usage, etc. in program binary code.

However there are still some fundamental issues coming from aspects of code static analysis with symbolic computations methods:

- cycle existence – amount of iteration calculation is not possible to be found with static analysis;
- exponential increase of analyzed path amount due to conditional branch number;
- there are debugging execution branches created by compiler rather than programmer;
- complexity of system calls emulation and library procedure semantics;
- limited prospects of automatically theorem proving system, complexity of gotten symbolic values condition systems.

Some of these issues can be partially solved by modern large-scale concolic approaches (the previously mentioned SAGE), or selective paths processing (S2E platform). However, today it seems unsolvable to deal with “exponential explosion” problem in general case.

REFERENCES

[1] Pechenkin, A.I., Lavrova, D.S. Modeling the search for vulnerabilities via the fuzzing method using an automation representation of network protocols. Aut. Control Comp. Sci. (2015) 49: 826. DOI: https://doi.org/10.3103/S0146411615080325.

[2] Pechenkin, A.I., Nikolskiy, A.V. Architecture of a scalable system of fuzzing network protocols on a multiprocessor cluster. Aut. Control Comp. Sci. (2015) 49: 758. DOI: https://doi.org/10.3103/S0146411615080313.

[3] Fuzzing @ Microsoft – A Research Perspective, Patrice Godefroid, Microsoft Research, ACSC 2017.

[4] Robert S. Boyer, Bernard Elspas, Karl N. Levitt SELECT--a formal system for testing and debugging programs by symbolic execution, Proceedings of the International Conference on Reliable Software, 1975, page 234--245, Los Angeles, California. DOI: https://doi.org/10.1145/800027.808445.

[5] James C. King, Symbolic Execution and Program Testing. Communications of the ACM, Vol. 19, Num. 7, 1976. DOI: https://doi.org/10.1145/360248.360252.

[6] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song. Loop-Extended Symbolic Execution on Binary Programs. In the Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), July 2009. DOI: https://doi.org/10.1145/1572272.1572299.

[7] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea. Selective Symbolic Execution. Appears in Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), Lisbon, Portugal, June 2009.

[8] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, Driller: Augmenting Fuzzing Through Selective Symbolic Execution. NDSS’16, 21-24 February 2016, San Diego, CA, USA. DOI: https://doi.org/10.14722/ndss.2016.23368.

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. Carnegie Mellon University Pittsburgh. May, 2016. PA. DOI: https://doi.org/10.1109/SP.2012.31