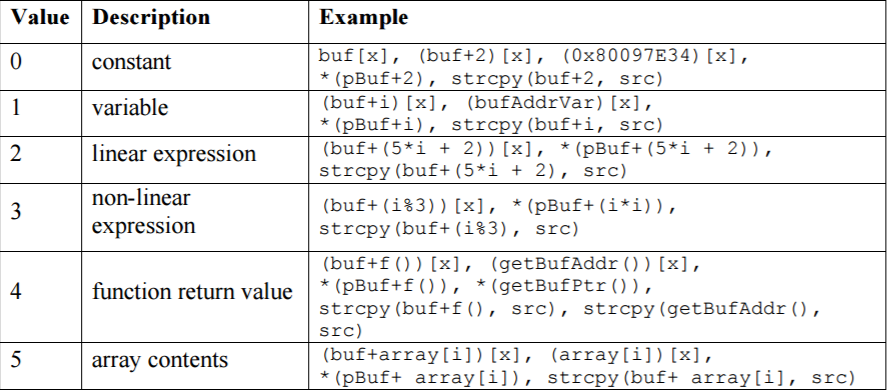
Project Report

Objective: Devise methods to find code that is likely to contain buffer overflow in C/C++ programs employ complexity analysis.

In computer security, buffer overflow is an anomalous situation where the data is written beyond the acceptable limit of the given storage (buffer). The problem is to figure out a method to identify code with bugs such as buffer overflows. Current researches done include Finding Buffer Overflow Inducing Loops in Binary Executables and Statistical Debugging Using a Hierarchical Model of Correlated Predicates. A general practice to find Buffer Overflow is to look for the presence of certain functions that manipulate string buffers, like the strcpy family. In these functions, data is moved from one buffer to another, within a loop, without considering destination buffer size. The goal of statistical debugging is to find faults in a program by identifying the predicates with the greatest deltas between the predicates in a correct and incorrect program execution. The predicates with the highest percentage delta between the two programs will point to the fault in the test program.

My initial approach to this problem was to analyze the set of code to find the Cyclomatic complexity of the code. Initial step is to determine the complexity of the code in compiled languages as they’re more prone to buffer overflows. I had configured GNU Complexity Measurement tool of C source for this approach. This tool uses multiple factors such as code length, Switch Statements, Logic Conditions. The complexity computation is done by counting lines of non-comment source lines, multiplies by a “nesting factor” for each level nesting and divides by a scaling factor so that the typical results lie roughly in the same range as pmccabe results. The complex set of code is then analyzed to find bugs using statistical debugging. Other tools such as Lizard and CCCC was employed to determine varies types of complexity. Lizard is an extensible Cyclomatic Complexity Analyzer for many imperative programming languages including C/C++. It counts the nloc (lines of code without comments), CCN (cyclomatic complexity number), token count of functions and parameter count of functions. This tool actually calculates how complex the code 'looks' rather than how complex the code really 'is'.

But over time my approach has changed from using Cyclomatic complexity based on the researches I’ve done. “A Taxonomy of Buffer Overflows for Evaluating Static and Dynamic Software Testing Tools” was the key papers I used for my project. To get a better understanding of the problem, I made an analysis of the different ways a buffer overflow could take place. The methods were selected in relation to the complexity of the code. I tried different softwares that that has been previously built to analyze and predict buffer overflows, such as BOON and SPLINT. Both these softwares were open source and had documentation that helped me work with it. BOON’s analysis  is  flow­ insensitive  and  context­ insensitive  for scalability and  simplicity. It focuses  exclusively on  the  misuse  of string manipulation  functions. It is a tool for automatically finding buffer overrun vulnerabilities in C source code. SPLINT employs “lightweight” static analysis and heuristics that are practical. The different methods used for buffer overflow detection were Index complexity, Address Complexity, Length/Limit Complexity and Loop Complexity. Index Complexity describes the complexity of the array index, if any, of the buffer access causing the overflow. Address complexity describes the complexity of the address  or pointer computation  (constant, variable, linear expression, non­linear expression, function  return  value, array contents). The Length Complexity attribute describes the complexity of the length or limit passed to the C library function, if any, that overflows the buffer. Loop Complexity describes what kind  of program control flow most immediately surrounds or affects the overflow (none, if, switch, cond, goto/label, setjmp/longjmp, function  pointer, recursion). For the  values “if”, “switch”, and  “cond”, the  buffer overflow is  located  within  the  conditional construct. I have focused my attention to dive deeper into address complexity. The Address Complexity attribute poses the question, “How complex is the address or pointer computation, if any, of the buffer being overflowed?” This attribute is used to describe the user program only, and is not applied to C library function internals. Here are some sample code that depicts the complexity of addresses:



My approach to this problem was to translate the C/C++ program in to assembly MIPS code. I used the online tool compiler explorer-C++. It translates C/C++ code into x86-64 gcc 4.4.7 assembly code. I then attempted to use the translated code and analyzed it to identify potential areas buffer-overflows can occur. The complexity was determined by the different types of values that were passed in to the buffer such constant, variable, linear expression, non-linear expression, function return value and array contents. The complexity of the program was determined based on the description the values with constant as the lowest complexity and array contents as the highest complexity. The higher the appearance of assembly code similar to “mov eax, DWORD PTR [rbp-40]” with a high offset of ‘40’ translated from “gets(buff+k[10]);” with ‘k[10]= 0x80097E3’ in C++ in areas surrounding the buffer indicated highest complexity thereby higher chances of buffer overflow. On the other hand, appearances of assemble code such as “mov eax, DWORD PTR [rbp-4]” with a significant lower offset ‘4’ translated from “gets(buff+j);” with ‘int j = 0x80097E3’ in C++ indicated lowest complexity thereby lower chances of buffer overflow occurrences. This study is still incomplete and requires further analysis for implementation.

Datasets were obtained from Github using the search for buffer overflow fix commits. Currently I’ve a semi functioning python script that scrapes code with the commit URL as input. I’m also working on identifying different libraries in C/C++ that leads to buffer overflows.

References

Rawat, S., & Mounier, L. (2012). Finding Buffer Overflow Inducing Loops in Binary Executables. 2012 IEEE Sixth International Conference on Software Security and Reliability. doi:10.1109/sere.2012.30

Parsa, S., Asadi-Aghbolaghi, M., & Vahidi-Asl, M. (2011). Statistical Debugging Using a Hierarchical Model of Correlated Predicates. Artificial Intelligence and Computational Intelligence Lecture Notes in Computer Science, 251-256. doi:10.1007/978-3-642-23881-9\_32

Kratkiewicz, K. (n.d.). A Taxonomy of Buffer Overflows for Evaluating Static and Dynamic Software Testing Tools. Retrieved from <https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full_papers/061107_Kratkiewicz.pdf>.

Kratkiewicz, K. J. (march 2005). Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. Retrieved from https://www.ll.mit.edu/mission/cybersec/publications/publication-files/full\_papers/KratkiewiczThesis.pdf.