Integrating static and dynamic analysis for detecting vulnerabilities

Ashish Aggarwal, Pankaj Jalote
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur, India 208016
{ashish.agg, jalote}@gmail.com

Abstract

A secure software demands effective techniques for vulnerability detection during its development cycle. The practice of detecting security flaws before the deployment phase eliminates the risks that vulnerabilities may impose for the company. Static analysis and dynamic analysis techniques offer two complimentary approaches for checking vulnerabilities. Static analysis involves the scanning of source code or binary eliminating the need of executing it. This approach is fast and has no run time overhead. However, static analysis are quite imprecise and generate huge false positives and false negatives. On the other hand, dynamic analysis involves the running of the software. The problem of false positives and negatives is less in case of dynamic analysis because they analyze by running the test cases. But this approach requires large number of test cases to ensure a certain confidence level in detecting security bugs. This paper describes a methodology which integrates the two approaches in a complimentary manner. It adopts the strengths of the two and eliminates their weaknesses. We are currently dealing with buffer overflow vulnerability with pointer aliasing. However the idea can be extended to other vulnerabilities also for e.g memory related errors, race conditions(Time of Check to Time to Use vulnerability), dangling pointer vulnerability, integer errors etc.

1 Introduction

"...There are too often software problems that prevent the end users from being able to improve cyber security by themselves. It is critical that we have the software companies fighting alongside us by improving the security quality of their products."- C. Micheal Armstrong, Chairman of Comcast and chairman of the Business Roundtable's Security Task Force,May 2004.

Industry experts claim that 70% of breaching attempts are made at application level. The problem of insecurity of a software is present everywhere - from commercial softwares to open source. Hence there is a great demand for detecting security flaws in the source code earlier in the development cycle. Analyzing the source code for security bugs before their deployment can help businesses reduce the risks. "The most cost effective actions that enterprises can take to increase IT security are to buy and build secure software. This is not that impossible, and not even that expensive, compared to fixing broken software after it's operational"-John Pescatore, Gartner, Inc.

In IT community, static and dynamic analysis are two complimentary approaches adopted to detect the security bugs. Dynamic analysis involves the actual running of the software whereas static analysis involves the scanning of source code or binary without running it. Various tools are available in the market for static and dynamic analysis and have been successful to some extent. But both of the approaches have their pros as well as cons.

Static analyzers are fast and simple to use. Since static analysis works on source code and can test code without running it, it can be used very effectively as a part of daily workflow during the development process. Thus it helps in identifying the problems much earlier when the cost of fixing the bugs is low. On the negative sides, static analyzers are not strong enough in analyzing and generate lots of false positives and false negatives. for e.g Splint [1] by Larochelle and Evans, is a lightweight static analyzer which analyzes the source code on the basis of annotations. It covers different security vulnerabilities, but generates a number of false positives and negatives. Non-annotated source code analyzers include BOON[5], Prefix[2], Prefast[4], ITS4[6] etc. These analysis tools do not require annotation methods to analyze the code, instead they build a model of the program execution and try to reduce the program to a simpler system to understand. for instance 'BOON' models the buffer overflow vulnerability as a set of integer constraint problem. Still these analyzers are not accurate enough and generate huge false positives and negatives.

Dynamic analyzers, on the other hand, are accurate and

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

generate less false positives. However they suffer from run time overhead and require a large test cases to ensure a certain confidence level in detecting security bugs. Since dynamic analysis tests an application with test suites, it is difficult to test all the conditions as a result they lack depth. Dynamic analyzers instrument the code to produce information regarding execution paths. If a vulnerability is found, a warning is written in a trace file. Tools like Purify[8], STOBO[7] etc. follow this approach for handling different kinds of vulnerabilities. Purify[8] is a testing tool to detect the memory related errors such as memory leaks. FIST[9] by Ghosh et. al., is based on fault injection technique where a number of errors are injected during the execution of the program and the responses are gathered in a log file. Though they perform more exact checking than can be done statically, but might miss some errors because some execution paths might not have been executed during testing.
Since the two approaches have plus as well as minus points, we need an integrated approach which employs both of the ideas effectively. It should adopt the strengths of the two and eliminate their weaknesses.

In this paper we have designed a model which handles the buffer overflow vulnerability with great precision and coverage by employing both static and dynamic analysis. Buffer overflow vulnerabilities are one of the most common security flaws. The known internet worm of 1988 was also based on the buffer overflow vulnerability in the fingerd server program. It occurs when a programmer does not perform bound checking while writing the data in a buffer or does incorrect bound checking(off-by-one error). In a typical case of buffer overflow problem, the attacker overfills the buffer with malicious code and changes the return address such that the control gets transferred to the contents present in the buffer. Stack based, heap based and pointer based buffer overflow problems are well documented in[10]. Static analysis tools available identify the buffer overflow problems by scanning the source code. However there is a trade off between their scalability and precision. In order to be scalable, they are sometimes imprecise and generate huge false positives and false negatives. On the other hand dynamic analyzers can handle the problem with higher precision but requires huge set of test cases to cover all the paths.

Our proposed approach which employs both these ideas together will have two benefits: firstly it reduces the number of test cases needed by dynamic analyzer to test the code for security bugs and secondly it will report more bugs with higher degree of precision as compared to static or dynamic analysis alone. We will describe our Workflow model and our approach in section II and III respectively. The implementation and working of tool is described in Section IV and V. Experimental results are shown in Section VI where we have compared our integrated approach with static/dynamic analysis alone.

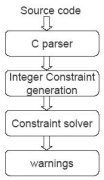

Figure 1. Architecture[5] of BOON - static analyzer

2 WorkFlow Model

This section onwards we will explain our approach to integrate the static and dynamic analysis effectively. We have used 'BOON'[5] as our static analyzer and 'STOBO'[7] as our dynamic analyzer for the experiments.

2.1 Overview of 'BOON'

'Boon'[5] is a static analyzer which formulates the buffer overflow detection problem as an integer constraint problem and uses simple graph theoretical knowledge for solving the constraints. It focuses more on scalability than precision, as a result it may generate false alarms(false positives) or may miss some of the bugs(false negatives). The architectural view of the 'BOON' is shown in Fig.1[5]
C parser parses the source code and models the buffers as pair of integer ranges - the number of bytes allocated to string and number of bytes actually used. The tool models the buffer overflow as integer range problem and perform integer range checking. For each string buffer it checks whether the allocated size is at least as large as its allocated length. For various C codes, it generates its own interpretation and checks for buffer overflow by integer range analysis. It has described its own constraint language to model the string operations. for e.g.
char s[n] implies size allocated to buffer 's' = n
strcpy(dst,src) implies len(src) = len(dst)
To ensure the security, it defines the safety property as len(s) should be less than alloc(s). Though the BOON tool is fast and scalable, yet it is not precise enough to cover all the security bugs. It does not handle the pointer aliasing efficiently as a result some of the buffer overruns in the source code are left unreported.

2.2 Overview of 'STOBO'

STOBO(Systematic Testing of Buffer Overflows)[7] is a dynamic analyzer based on the testing technique. It instru-

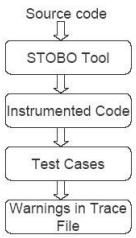

Figure 2. Architecture of STOBO - dynamic analyzer

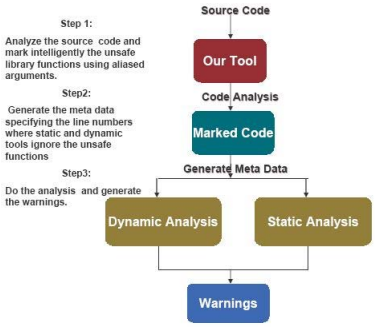

Figure 3. WorkFlow of our Model

ments the source code with small snippets which tracks the buffers and checks library functions for certain safety property. Library functions have preconditions for e.g. in strcpy function - the allocated memory to destination buffer should be greater than the length of the string in the source buffer. If such conditions are violated the buffer overflow occurs and the warnings are reported in a trace file. The architectural view of the STOBO is shown in Fig.2. STOBO[7] takes source file as input and generates a new file with code instrumented. When executed,it has the same behavior but the information regarding the testing coverage is generated in a trace file. It focuses on the unsafe library function-'strcpy'. It replaces it with a wrapper function __STOBO_strcpy() which checks for preconditions before calling the actual library function and reports any overflow if occurred during the testing phase. However, the main problem associated with dynamic analysis is the number of test cases to ensure a certain confidence level in detecting security flaws.

3 Integrating static and dynamic Analysis

Our model [Fig.3]makes use of static and dynamic analysis together in an effective manner with the help of a tool. Consider the following C code. Suppose there are two components which provide different functionalities in an application.

```
1.  component1(int n){
2.  char tmp1[10], *p;
3.  strcpy(tmp1,"Welcome");
4.  p = tmp1 + n;
5.  strcpy(p," India");
6.  }
7.  component2(char *input){
8.  char tmp2[10];
9.  strcpy(tmp2,input);
10. }
```

This example illustrates the working and the benefits of our model. Static tools normally fail to detect buffer overflow vulnerability due to the pointer aliasing. In this example code, local pointer variable 'tmp1' forms an alias with 'p'(line number 5) in component1. Variable tmp1 has 10 bytes of memory allocated and its alias(variable 'p') points to only last (10-n) bytes of 'p'. The strcpy function at line 6 is unsafe because 6 bytes("India" + a nul string) are being copied in (10-n) bytes of memory, for n= 7, 6 bytes are written in 3 byte buffer. This buffer overflow vulnerability is undetected by static tools(false negative) due to pointer aliasing. However, in component2, where the strcpy function(at line 12) is also unsafe to use(since input string can be of any length), the vulnerability is detected by the static analyzer because there is no problem of pointer aliasing.

On the other hand a dynamic analyzer like STOBO will report both the bugs since it keeps certain information for each variable at run time like memory allocated to them, their starting address and length of data string written in the buffer. Hence while testing component1, dynamic analyzer will execute the function called __STOBO_strcpy(a wrapper of strcpy) which checks for safety properties. At run time, the dynamic analyzer will know that p points to only (10-n) bytes of memory where data can be written without any overflow at run time. Hence it raises an alarm when an attempt is made to overwrite the buffer(at line 6). While testing component2, the dynamic analyzer will again warn of a potential buffer overflow vulnerability at line 11 if the length of input argument exceeds 10 bytes(last byte is for nul string).

Dynamic analyzers are precise but they need test cases. The number of test cases required is huge when an application has large number of complex components. Here for component1 various values of 'n' will be tested to ensure that buffer overflow is not a problem and for component2 various input strings of different lengths will be used as test

cases. However we may need test cases only for component1 since the static analyzer fails to detect the possible vulnerability. The component2 can be analyzed by static analyzer with good precision and therefore it does not require dynamic analysis.

Our tool solves this problem by marking unsafe library functions(like strcpy) using aliased variables and guide the dynamic analyzer to focus only on them. Our tool will scan the source code for pointer aliasing. It marks the variables(C pointers) as 'suspicious variables' if they are doubted to be aliased. Unsafe library functions like strcpy etc. using these 'suspicious variables' as arguments are marked for dynamic analysis. Since aliasing is frequently found in C programs, our tool must be intelligent enough in marking only those functions which cannot be handled by static analyzers with precision. With the help of this tool, we logically partition the code into two - marked functions and unmarked functions. For marked functions to be handled by a dynamic analyzer, meta data is generated for the dynamic analysis tool, specifying the line numbers where it should focus for detecting the security flaws. Meta data is also generated for a static analyzer which is used for suppressing the spurious warnings generated by it. The number of false positives and false negatives gets reduced on the whole.

Hence our model uses the strengths of the two analysis approaches. Dynamic analysis, though accurate, but requires lots of test cases. In our model we need less test cases for dynamic analysis because the coverage area is reduced. Only suspicious functions are handled by the dynamic analyzer. On the other hand static analysis, though fast and has no runtime overhead, are not thorough enough in their analysis. Therefore those functions which can be handled by static analyzers with higher precision are passed to them for analysis. We will describe the working of our tool in following section.

4 Implementation of Tool

We will describe the problem and then the method to solve it in the following subsections.

4.1 Problem Description

An alias occurs in a program when two or more names exist for the same memory location during the program execution. C language allows arbitrary pointers, hence the problem of determining aliases at a program point is P-space hard. A highly precise algorithm for determining pointer aliasing will improve the performance of our model. It helps in marking less number of unsafe library functions that need to be handled by dynamic analyzer. As a result we will need few test cases and the run time overhead will be less. However an approximation algorithm will also work fine and generate adequate results. We have added some intelligence to the tool to improve the performance. There are cases where the pointers are aliased but are not unsafe to be used by library functions. William Landi at. el.[11] have devised a safe approximation algorithm for pointer aliasing. The researchers in this field have focused their attention on finding the possible aliases of a particular name at some program point 't'. All proposed algorithms try to find all variable names that refer to the same memory location on some path to 't'. Our algorithm works in parallel with the present work but we do not need the information of the variable names that are aliases of a particular variable. We are concerned whether any pointer is aliased at some program point 't', in other words, does there exist a path to 't' that results in aliasing of concerned variable name? Here the program point is the location in the code where unsafe library function(strcpy) is called and the arguments to these library functions are our concerned variables. The following definitions will be used throughout the paper:

Suspicious pointer: A pointer variable is said to be suspicious if there exists an alias of that pointer.
Marked function: Unsafe library functions(strcpy) are said to be marked if the arguments are aliased.
DereferencedToStore Pointer Variables: If the pointers are dereferenced to write at the memory location pointed by them, such pointers are said to be dereferncedToStore.

Thus if there exists any path to 'strcpy' function such that the arguments to it are suspicious then we will mark them for dynamic analysis. However this scheme marks majority of the library functions but they can be handled with good precision by static analyzers. for instance if a particular pointer variable 'p' is aliased with some name 'q', but 'q' is not DereferencedToStore, then p is safe to be handled by static analyzer.
Consider the following C code for detailed analysis:

```
P()
{
char * var = (char *) malloc(10);
strcpy(var,"XYZ"); //var is not aliased
MAKE_DIR(var);
strcpy(var,"ABC"); //var is aliased but not unsafe
}
MAKE_DIR(char *name)
{
mkdir(name,"0"); //mkdir system call
}
```

In the above code snippet, pointer variable 'var' is aliased with 'name' when P() calls function make_dir(). Since 'var' is not aliased before the MAKE_DIR function call is made, strcpy function at line number 3 is safe to use. But after the MAKE_DIR procedure call, the strcpy

function at line 7 is treated with suspicion. However if we look carefully, we find that MAKE_DIR() function does not change the contents of the memory location pointed by 'var'. It dereferences it to retrieve the directory name. Hence the strcpy function at line 7 can be left for analysis by a static analyzer.

4.2 Program Modeling

We represent the program by a Control Flow Graph which intuitively is the union of statement level control flow graphs for each procedure augmented by call nodes. Call nodes represent the functions that are invoked by any procedure. The model will consists of only those nodes which use the pointer variables. In order to understand the working of the tool, we divide this section into 3 subsections. In first subsection we will see how to deal with the local pointers of a procedure, in next subsection we will deal with pointer arguments that are passed to a procedure and in final subsection we will look into the global pointer variables.

For every procedure we maintain a list of the local variables and arguments. Each variable name(local or argument) has a tag associated with it which determines its suspicion level. All local variables are initiated with suspicion level '0'. A local variable can have only two suspicion levels '0' or '1'. The suspicion level of the arguments is decided by the caller of that procedure. It can have three suspicion levels '0', '1' or '2'.
We scan each statement in a function for the possibility of aliasing. If any variable name is found to be aliased with some other variable name, we mark both the variables with some suspicion level. We will look for the following patterns to decide the suspicion level.

1. Case 1: q = Some Function involving 'p' and integer values
2. Case 2: ProcedureCall(p,....)
3. Case 3: ProcedureCall((Some function of 'p' and integer values),....)
4. Case 4: 'p' = ProcedureCall(...)

In order to understand how the suspicion level is decided, we divide this section into 3 subsections. In first subsection we will see how to deal with the local pointer variables of a procedure, in next subsection we will deal with arguments that are passed to a procedure and in final subsection we will look into the global pointer variables.

4.3 Local Variables

We will now consider all the four cases for a local variable 'p' and 'q'.

Case 1 : We will set the suspicion level of 'p' and 'q' as '1'
Case 2 : If 'p' has aliases(suspicion level is '1'), then the arguments of the procedure that is being invoked (here 'ProcedureCall') are marked with suspicion level '2'. However if 'p' has no aliases, then the arguments of the procedure are marked with suspicion level '0'. We will update the suspicion level of the variable 'p' when the invoked procedure returns. Each variable passed as an argument ('p' in this case) maintains a special tag with values '0'(NO RISK) or '1'(COMPLETE RISK). This tag value is set when the called procedure returns. We will show how this value is set in our next subsection. So in this case, if the variable 'p' returns with value 'COMPLETE RISK' we will set its suspicion level '1' otherwise it is maintained at its original level.
Case 3 : If 'p' has no aliases, then the arguments of the procedure that is being called(here 'ProcedureCall') are marked with suspicion level '1'. However if 'p' has aliases then the arguments of the procedure are marked with suspicion level '2'. The suspicion level of the variable 'p' passed as an argument is updated(as described in case 2)when the procedure, being called, returns.
Case 4 : We will mark 'p' with the same suspicion level that is maintained by the pointer variable returned by the procedure that is being called.

4.4 Argument Variables

We will now consider all the four cases for a argument variable 'p'
Case 1 : We will set the suspicion level of 'p' as '2'.
Case 2 : The arguments of the procedure being invoked are set with the same tag value as maintained by 'p'. We will update the suspicion level of 'p' when the invoked function returns. If it returns with value 'COMPLETE RISK' we will set its suspicion level '2' otherwise it is maintained at its original level.
Case 3 : If the suspicion tag value of 'p' is '0' or '1', the arguments of the procedure being called are set with tag value '1'. However, if the suspicion tag value of 'p' is '2' the arguments are also marked with suspicion level '2'. We will update the suspicion level of 'p' when the invoked function returns. If it returns with value 'COMPLETE RISK' we will set its suspicion level '2' otherwise it is maintained at its original level.
Case 4 :If the pointer variable returned by the procedure has suspicion tag value greater than '0', then 'p' is set with suspicion level '2'.
If the argument is not DereferencedToStore and has suspicion level '0' or '1', the special tag value is set to 'NO RISK'. In all other cases the special tag value is set to 'COMPLETE RISK'.

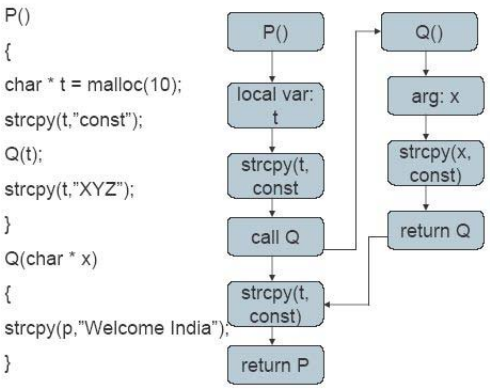

Figure 4. Example code 1 and its CFG

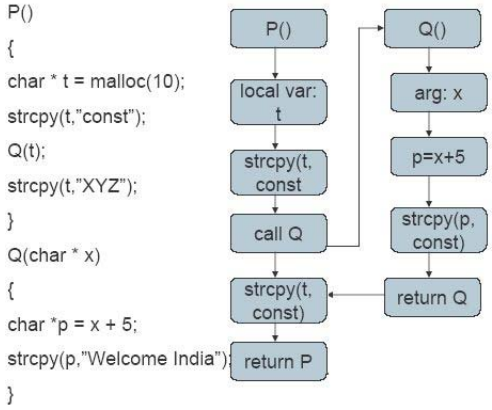

Figure 5. Example code 2 and its CFG

4.5 Global Variables

We will maintain the suspicion level of the global variables starting with zero level. However if there occurs aliases to global variables we will mark them as suspicious arguments with tag value '1'. We will treat global variables similar to the local variables. But the difference is that the suspicion level of global variables is not initialized to '0' in each function call.

5 Marking Unsafe Library functions for Dynamic Analysis

If the arguments to the 'strcpy library function' are of type 'local or global' having nonzero suspicion level, or of type 'arg', having high suspicion level only, then we will mark them for dynamic analysis. We will look into some examples to understand how the tool actually works.

5.1 Example 1

Consider the code and its CFG in Fig.4 The procedure 'P' has a local variables 't' which starts with zero level of suspicion. At line 5, P() calls Q(). The argument(variable name 'x') of Q is marked with type 'arg' and suspicion level as '0'. The strcpy function in procedure Q() is safe and hence not marked because the argument to strcpy function is at zero level suspicion. The function returns with value 0('NO RISK') according to the set of rules stated above. Hence the suspicion level of variable 't' is kept at '0'. The strcpy function at line 6 is not marked since its arguments are not suspicious.

5.2 Example 2

Consider the code and its CFG in Fig.5 The procedure 'P' has a local variable with zero level of suspicion. At line 5, P calls Q. The argument is marked with zero level suspicion. But the aliasing within Q (at line 10) raises the suspicion level of 'p' and and 'x' and hence the strcpy function is marked for dynamic analysis. The function 'Q' returns with tag value '1'(COMPELETE RISK), hence the suspicion level of local variable 't' is set to '1' and strcpy function within P is also marked for dynamic analysis.

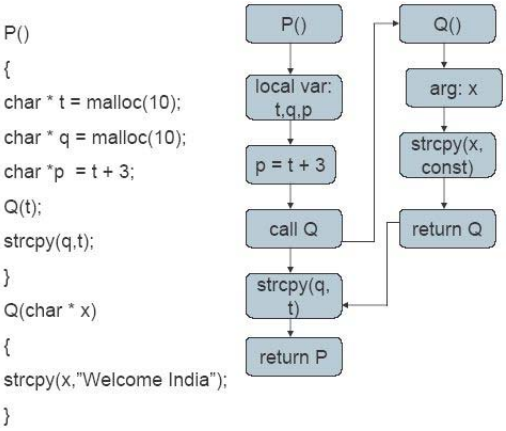

Figure 6. Example code 3 and its CFG

5.3 Example 3

Consider the code and its CFG in Fig.6 The procedure P has three local variables 'p', 'q' and 't' which start with zero level of suspicion. At line number 5, 'p' and 't' are aliased and are marked with suspicion tag value '1'. When P() calls Q(), the argument of Q(named 'x) is marked with suspicion level '2'. The strcpy function called in procedure Q() is marked for dynamic analysis because 'x' is of type 'arg' and has suspicion level greater than zero. The strcpy function at line 7 is also marked for dynamic analysis since

one of its arguments('t') is marked suspicious.

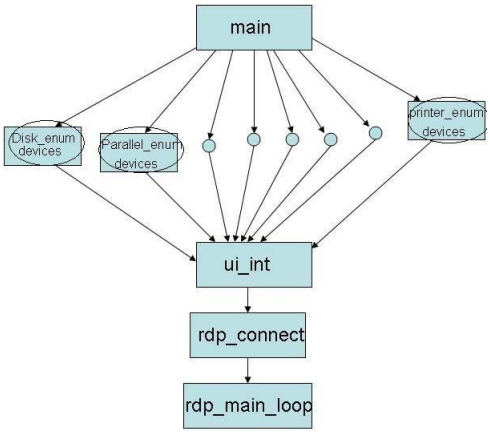

Figure 7. Experimental Results

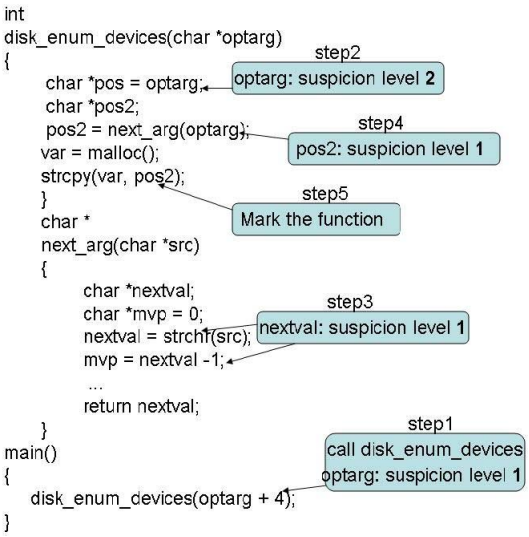

Figure 8. Marking of 'strcpy' function

6 Experiments

The latest version of rdesktop(about 17k lines of code)was one of the first programs analyzed. The sample output is shown in the form of control graph Fig.7 where the 'main' function follows one of the control path depending upon the arguments. All paths diverging from the 'main' function converge to the basic functions - ui_init followed by rdp_connect and rdp_main_loop. The encircled functions use the 'strcpy' with suspicious arguments, hence they are marked for dynamic analysis. The rest of the code is safe to be handled by the static analyzer. On manual analysis we find that the number of test cases has greatly reduced. We need to test only one option i-e. '-r' with the argument 'disk'+'some string'(for function - disk_enum_devices), 'printer'+'some string'(for function - printer_enum_devices) and 'lpport'+'some string'(for function - parallel_enum_devices). Since we do not need to check the remaining path from the encircled function to the end of the control graph, there is a significant reduction in the run time overhead. Let us give an example to demonstrate how our tool marks the strcpy function of disk_enum_devices for dynamic analysis. It is illustrated in Fig.8

On the other hand, the performance of the static analyzer has improved in terms of precision. The warnings generated by the static analysis tool are more accurate. In short our model shows two benefits: firstly it reduces the number of test cases needed by the dynamic analyzer and secondly it reported more bugs as compared to previous models.

We have also conducted our experiments on the latest version of 'wzdftpd' program. Out of 64 instances of strcpy functions only 24 instances are marked for dynamic analysis. On manual auditing we find that the number of components that need dynamic analysis is greatly reduced. The functionalities - 'do_mkdir, do_rmdir, do_list, do_help and many others can be handled by static analyzers with great precision since they use safe(unmarked) strcpy functions. The number of test cases required for dynamic analysis reduces, since the number of components to be tested has decreased. We will demonstrate this with an example of 'do_mkdir' which does not require dynamic analysis. Consider the call graph of the do_mkdir function in Fig.9.

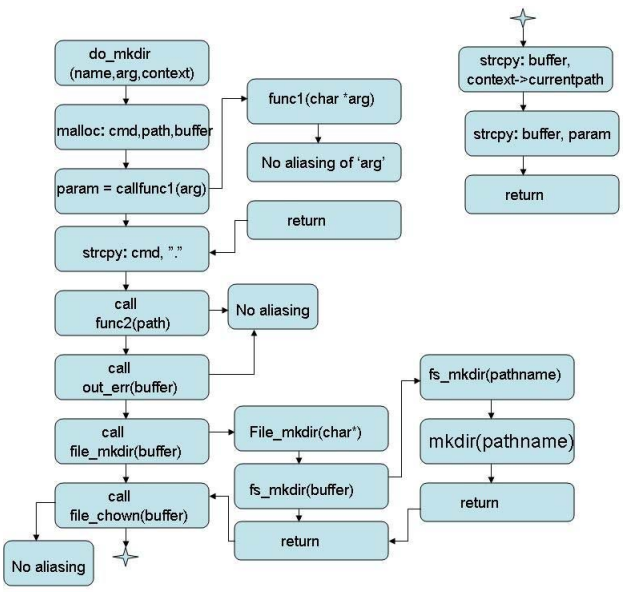

Figure 9. Call Graph: 'do_mkdir'

The arguments to the strcpy function in the graph Fig.9 are

non suspicious. Hence they are not marked for the dynamic analysis. Similarly other functions are also found safe to be handled by static analyzers.

7 Limitations

The performance of our model depends upon the smartness of the tool in marking the 'strcpy library function'. C language is wide enough and has various constructs like struct and unions. Presently our tool does not handle the aliasing of the structure elements(struct) intelligently. By manual auditing we find that it misses some of the strcpy functions involving the aliased structure elements. We need to improve the tool so that it can handle the C constructs with an acceptable precision. However, the results have been quite satisfactory for the rest.

8 Conclusion

The two approaches - dynamic analysis and static analysis - complement each other and play a significant role in detecting the security vulnerabilities. But the right combination of the two approaches can enhance the security level of an application by detecting more flaws and with higher precision. Static tools are generally imprecise and generate false positives and false negatives whereas dynamic tools, though precise, requires huge set of test cases. We have tried to integrate the two approaches effectively. With the help of a tool we mark the unsafe library functions(strcpy) using aliased arguments for dynamic analysis because such functions are not handled by static analyzers with good precision. Since the coverage area for dynamic analysis gets reduced, the number of test cases required is less. In this manner we try to incorporate the benefits of the two analysis eliminating their cons to some extent. In our paper we have focused only on the buffer overflow vulnerability, still the idea of integrating the two analysis approaches can be used for detecting other vulnerabilities also like race conditions, integer errors etc.

References

[1] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, pages 177190, Washington, District of Columbia, U.S.A., August 2001. USENIX Association.

[2] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. Software: Practice and Experience, 30(7):775802, June 2000. ISSN: 0038-0644.

[3] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In Proceedings of the Winter 92 USENIX conference, pages 125136, San Francisco, California, U.S.A., January 1992. USENIX Association.

[4] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting Software. IEEE Software, 21(3):92100, May-June 2004.

[5] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Proceedings of the 7th Networking and Distributed System Security Symposium 2000, pages 317, San Diego, California, U.S.A., February 2000.

[6] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In 16th Annual Computer Security Applications Conference, New Orleans, Louisiana, U.S.A., December 2000.

[7] Eric Haugh and Matt Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In Proceedings of the 10th Network and Distributed System Security Symposium (NDSS03), San Diego, California, U.S.A., February 2003. Internet Society.

[8] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In Proceedings of the Winter 92 USENIX conference, pages 125136, San Francisco, California, U.S.A., January 1992. USENIX Association.

[9] Anup K. Ghosh, Tom OConnor, and Garry McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In Proceedings of the IEEE Symposium on Security and Privacy, pages 104114, Oakland, California, U.S.A., May 1998. IEEE Computer Society, IEEE Press.

[10] Yves Younan, Wouter Joosen, Frank Piessens. Code Injection in C and C++ : A Survey of Vulnerabilities and Countermeasures, Report CW386, July 2004, Department of Computer Science, K.U.Leuven

[11] William Landi, Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing, Department of Computer Science, Rutgers University, New Brunswick, NJ