Hybrid Analysis of Executables to Detect Security Vulnerabilities

Pranith Kumar D
Dept. Computer Sc. & Engg.
IIT Kharagpur
Kharagpur, WB 721302, India
Email: bobby.prani@gmail.com

Anchal Nema
Dept. Computer Sc. & Engg.
IIT Kharagpur
Kharagpur, WB 721302, India
Email: anchaliitkgp@gmail.com

Rajeev Kumar
Dept. Computer Sc. & Engg.
IIT Kharagpur
Kharagpur, WB 721302, India
Email: rkumar@cse.iitkgp.ernet.in

Abstract—Detection of vulnerabilities in executables is one of the major problems facing the software industry. Cracking has increased the complexity where even the programs from reputed companies might be made malicious if they are vulnerable. The main challenge in analysis of executables is due to the unavailability of the source code. Results generated using dynamic analysis alone are unsound, i.e., they do no generalize. On the other hand, results generated using static analysis are usually conservative and it will report weaker properties which might be true but useless. Also, using a model in static analysis which is complete, will render the analysis complex and slow. In this work, we present a hybrid approach, which uses a combination of static and dynamic analysis to identify vulnerabilities. In this approach, we first instrument the executable to extract the control flow of the program. We get the assembly code of the instructions being executed. We then perform static analysis on the assembly code using control flow and register bounds. Static analysis uses slicing on the disassembled code and constraint bound checking is performed on the slice generated. In this way, we exploit the synergy between static and dynamic analysis. Memory errors including memory leaks, buffer overflow and dangling pointers found during the hybrid analysis are reported.

Index Terms—Hybrid analysis, security vulnerabilities, malicious code, memory errors, instrumentation, slicing, validation.

I. INTRODUCTION

Errors introduced by programmers can also be exploited by malicious code writers. These errors can be logical or are introduced when programmers do not take sufficient care at the time of writing the code. For example, low level programming languages like C and C++ provide the flexibility for programmers to have arbitrary memory accesses within the programs address space in the form of pointers. When the programmers are not careful enough when using these features, it might result in illegal memory reads and writes. These illegal reads and writes are then exploited by malicious code writers to tamper with the system.

Buffer overflows accounts for approximately half of all security vulnerabilities [5] Simplest buffer overflow attack, stack smashing [2] overwrites a buffer on the stack to return the return address. When the function returns, control will jump to the address that was replaced on the top of the stack by the attacker which provides him the ability to execute arbitrary code. Many standard C library functions provides unsafe functions(such as gets) that writes an unbounded amount of user input into a fixed size buffer without any bounds checking which can be exploited by attackers. Further, dangling pointers can also be exploited in much the same way as buffer overruns to compromise system security[15]. Dangling pointers are created when a memory block pointed to by more than one pointer is freed using another pointer, whilst this pointer keeps pointing to the freed up space.

Researchers have devised various ways of doing analysis on source code as well as on executables. In presence of source code, it can be checked to follow coding standards listed in [7] to ensure that it is free from known security vulnerabilities like dangling pointers, buffer overflows and others. LCLint [11], a source code analyzer, uses the information provided in semantic comments embedded in source code to detect buffer overflow. In the absence of source code, the above methods cannot be used to detect security vulnerabilities.

Various static and dynamic analysis techniques on the binary code are adapted to detect security vulnerabilities. Both the static and dynamic techniques have their pros and cons. Static analysis is conservative giving rise to weaker properties which may not be useful whereas in dynamic analysis, results generated cannot be generalized for all possible inputs.

In our work, we present a hybrid approach which utilizes the strength of both dynamic and static analysis to efficiently detect security vulnerabilities like buffer overflow, dangling pointers and memory leaks. Executable is first instrumented using PIN to extract the exact control flow and register bounds. Executable is disassembled to get the assembly code. Control flow and register bounds are then used in static analysis in which constraint bound check is performed on the slice generated. Finally memory errors obtained as discussed earlier are reported.

Section II presents the previous work done in analysis, techniques and tools along with the motivation of the work. Section III-A discusses the overall architecture of the approach adopted to detect memory errors. Section III-B presents the static slicing used in the approach discussed in Subsection III-A. Section III-C provides the details of PIN and its applications in our analysis. Section IV provides an overview of type of memory errors and the approach adopted to detect them. Section IV-A presents the consequences of memory leaks and our method of detection with examples. Section
IV-B presents the hazards of buffer overflow and our method of detection with examples. Section IV-C presents the dangling pointers detection algorithm and the results obtained. Section V concludes the work presented in this paper.

II. MOTIVATION AND BACKGROUND

In the scenario where the source code is available, static analysis can audit the source code for checking the compliance with coding standards. Graff lists several secure coding practices in [7]. Several tools [13] have been developed which check the source code for known security vulnerabilities. These may include checks for buffer overflows, dangling pointers, use of uninitialized variables, etc. Most vulnerabilities are exploited through malicious I/O. So, validating the I/O data in the source code will prevent most of the exploits. This can also be done at runtime through data flow analysis in the absence of the source code [16]. BOON [18], CodeWizard [1], FlawFinder [20], [19] are a few examples of source code analyzers which create a database of general rules which affect the programs. They also check if the program uses any analyzers which create a database of general rules which affect the programs. If the program uses any vulnerable library functions like strcpy(), strcat(), gets() etc., LCLint [11] is another source code analysis tool which uses the information provided by the programmer in the semantic comments to detect likely buffer overflow vulnerabilities. This requires extra effort on part of the programmer which is not feasible for already written code. This kind of analysis is difficult in the absence of the source code. The binary code can be disassembled to get to assembly, but constructing a model for assembly with the same features as for a higher level language is difficult. Static analysis for binaries usually rely on flow-based methods. In this approach, they statically try to determine the flow of the program and data. Depending on the model constructed, this kind of analysis can be hugely complex [3].

The static analysis done by Tevis in [17] involves the scanning of executable files for software security vulnerabilities. The methodology uses the PE format (designed for software running on Windows NT/XP) and extracts information located in the headers, sections and tables of an executable files, along with the overall content of the file as a means to detect specific anomalies and security vulnerabilities without having to disassemble the code. Anomalies include inconsistent table sizes, zero filled regions of bytes, unknown region of bytes, compressed file placed in the file, sections that are both writable and executable, use of functions that are susceptible to buffer overflow attacks. This methodology extracts a very little information about the behavior of executable.

The work by Bergeron, in [4], uses static slicing of the disassembled code obtained from the executable. The static slicing is useful to extract the code fragment that are critical for the security of the system. Once these fragments are extracted, they are subjected to behavior specifications to decide whether they are malicious or not. The methodology is incapable to find dynamic aspects of the code which can only be found while executing the code.

Hybrid analysis attempts to erase the boundaries between static and dynamic analysis and create unified analyses that can operate in either mode or in a mode that utilizes the strength of both approaches. Static or dynamic analyses can enhance one another by providing information that would otherwise be unavailable. Hybrid analyses would sacrifice a small amount of the soundness of static analysis and a small amount of accuracy of dynamic analysis to obtain new techniques whose properties are better suited for particular cases than purely static or dynamic analysis [6].

Consider the C-code given in Fig. 6. Memory access out of bound (a[10]) occurs in it. Since the source code is generally unavailable we just have the disassembled code obtained from the executable shown in Fig. 7. Whenever a malloc() is called, we extract the start address and the size of the memory block using the memory profiler of PIN during the execution to get the upper and lower bounds of the allocated memory space. Therefore whenever a write to a memory block occurs, using static backward slicing, we determine the upper and lower bounds of the register involved in memory write. If the memory access is invalid, we report the memory access error. Upper and lower bounds of the allocated memory area can be easily found using dynamic analysis. Upper and lower bound of any register (related to memory write) can be found easily using static analysis as explained later in Subsection III-B. In this way, we efficiently use the strengths of both static and dynamic analysis to detect memory access errors.

III. HYBRID ANALYSIS

In this section, we discuss the methodology and the major steps of the analysis, namely static slicing and dynamic analysis, which we report in this work.

A. Methodology

As shown in Fig. 1, in the first step, the executable is dynamically analyzed using PIN tools to get the virtual address
of the instructions being executed. This virtual addresses are then mapped to the corresponding assembly code extracted using the objdump of the executable. This gives us the exact sequence of instructions being run on the system. In the second step, we instrument the code to make it analyzable. In the third step, we perform static slicing of this assembly code on constraints extracted during runtime to get a slice of code on which we run the program checker. Here we use hybrid analysis to get the more precise slice. Also we build up tables using dynamic analysis as we explain later and decide upon the contents of these tables if there are any errors present in the executable.

B. Static Slicing

One of the steps in the overall scheme is to slice the binary code to retain only the relevant set of instructions that are useful in analysis. There are basically two types of dependences between the instructions present in the program – Data dependence and Control dependence. Slicing algorithm uses both dependencies between the instructions present in the program. Given the pair \((s,V)\), called the slicing criterion, where \(s\) is the node in the control flow graph and \(V\) is the subset of program variables, it produces a set \(S_c\) of program instructions that are relevant for the computation of the variables in \(V\). The set \(S_c\) is called a slice. There are two types of slices that can be computed – Forward Slice and Backward Slice. A forward Slice to a program with respect to the slicing criterion \(s,V\) at a program point \(p\) consists of statements and predicates in the program that may be affected by the variables in \(V\) at \(p\). Similarly, a backward slice of a program with respect to the slicing criterion at a program point \(p\) consists of all statements and predicates in the program that may affect the value of variables in \(V\) at \(p\).

The algorithm presented in this section uses Procedure dependence graph (PDG). PDG is constructed using Control Flow Graph (CFG) by adding new data-dependent and control-dependent edges between the nodes of the program. Each node in the PDG represents the different instruction and each edge is either a control dependent edge or a data dependent edge from node \(i\) to node \(j\) for \(i, j\) belong to the node of the program. CFG is constructed using PIN instrumentation and is discussed later in this paper.

Intuitively, a statement \(j\) is data dependent on statement \(i\) if a value computed at \(i\) is used at \(j\) in some program execution. Control dependence is defined in terms of post dominance. A node \(i\) in the CFG is post-dominated by a node by \(j\) if all paths from it to end pass through \(j\). A node \(j\) is control dependent on a node \(i\) if(i)there exists a path \(P\) from \(i\) to \(j\) such that \(j\) post-dominates every node in \(P\), excluding \(i\) and \(j\), and(ii) is not post-dominated by \(j\).

In this work we show analysis on the PDG of the program. This work can be extended to System Dependence Graph (SDG) formed by the collection of PDGs. For vertex \(s\) of PDG \(G\), the slice of \(G\) with respect to \(s\), denoted by slice\((G,s)\) are all the set of nodes reachable from \(s\) in the PDG. Thus, slicing a program can be viewed as a graph reachability problem [8] in the Procedure Dependence Graph.

C. Dynamic Analysis

Dynamic analysis is the method of analyzing the properties of a running program. In contrast to static analysis, where we examine a program’s text to derive properties which hold for all possible executions, in dynamic analysis we extract properties which may hold for only that particular execution. The advantage here being that, the properties extracted are accurate. This kind of analysis is usually done through instrumentation [12]. Dynamic analysis is helpful in detecting violations of certain properties and also provides useful information to programmers about the behavior of their programs.

We specifically use itrace and malloc. itrace extracts the virtual addresses of the instructions being executed and malloc extracts the start address and size of the dynamic memory blocks allocated. We use PIN instrumentation library [9] to perform dynamic analysis on the executable. PIN allows us to instrument the executable by placing calls at arbitrary locations. It provides an easy to use and rich sets of APIs. It includes several sample architecture independent Pintools like profilers, trace analyzers. We use the instruction trace tool and the memory profiling tool [9].

In the instruction trace tool we extract the virtual addresses of all the instructions being executed. We now map these addresses to the addresses in the disassembly of the executable and extract the corresponding instruction being executed. (The disassembly is extracted by using generic tools like objdump). By this method, we get the exact control flow of the program. The CFG so constructed is efficient since the set of possible targets of indirect jump sites is reduced by using dynamic information similar to the work in [10]. A loop running for five times will have all its instructions appear five times in this flow. Also, we use a memory profiling tool malloc which gives us the address of a memory block and the size of the memory block during allocation time. We use these to form the Allocated Memory Table (AMT) the contents of which are explained in the next section.

IV. DETECTION OF MEMORY ACCESS ERRORS

In this paper, we concentrate on finding memory errors which are the main source of vulnerabilities in an executable, because of which programmers spend much time looking for these errors, but end-users may experience them first. The survey in [14] empirically shows the continuing prevalence of access errors in many widely used UNIX programs.

We maintain an Allocated Memory Table (AMT) which stores the various information regarding a given block. The information pertains to the start address of a memory block, the size of the memory block. These together give us both upper bound and lower bound address for that memory block. We also store a list of pointers pointing to this memory block which is calculated using the slicing algorithm. There is also a parent section for each memory block which helps us to keep
track of memory blocks which point to other memory blocks. A memory block which points to another memory block is its parent. There is also an active field which indicates whether an allocated memory block is active, i.e., in use or has been freed (inactive). The exact use of these fields is illustrated with examples in the Application section.

Memory access errors are typically classified into three types. They are — memory leaks, array index out of bounds (buffer overflows) and dangling pointers.

In this paper, we concentrate on finding such memory errors which are the main source of vulnerabilities in an executable.

A. Memory Leaks

Memory leak is a scenario in which memory allocated to a program is no longer accessible to the program due to some logic flaw on the part of the programmer. Memory leaks are more difficult to detect than other memory access errors. Memory leaks occur because a block of memory was not freed, and hence are errors of omission, rather than commission. In addition, memory leaks rarely produce directly observable errors, but instead cumulatively degrade overall performance.

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down unacceptably due to thrashing.

1) Challenges: Once found, memory leaks remain challenging to fix. If memory is freed prematurely, memory access errors can result. Since access errors can introduce intermittent problems, memory leak fixes may require lengthy testing. Often, complicated memory ownership protocols are required to administer dynamic memory.

Incorrectly coded boundary cases can lurk in otherwise stable code for years. Both memory leaks and access errors are easy to introduce into a program but hard to eliminate. Without facilities for detecting memory access errors, it is risky for programmers to attempt to reclaim leaked memory aggressively because that may introduce freed-memory access errors with unpredictable results.

Conversely, without feedback on memory leaks, programmers may waste memory by minimizing free calls in order to avoid freed-memory access errors. A facility that reported on both a program’s memory access errors and its memory leaks could greatly benefit developers by improving the robustness and performance of their programs.

2) Detection of Memory leaks: We consider the following two cases of memory leaks:

Case 1. When the last Reference to the memory block is overwritten:

Consider the code snippet given in Fig. 2 for Case 1. All the C codes including Fig. 2 are presented along with the assembly code for the better understanding of the latter. However, these codes are not used, since the analysis is done on the disassembled code obtained from the executable assuming the absence of the source code.

```c
void mem_leak1(void)
{
    char *p;
    p = (char *)malloc(10 * sizeof(char)); // last reference lost
    p = (char *)malloc(10 * sizeof(char));
    return;
}
```

The corresponding assembly code extracted using PIN is given in Fig. 3.

```assembly
mem_leak1:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $10, (%esp)
    call malloc
    movl %eax, -4(%ebp)
    movl $10, (%esp)
    call malloc
    movl %eax, -4(%ebp)
    leave
    end
```

Fig. 2. Memory leak example C code

During program execution we dynamically construct the AMT (Allocated Memory Table) as explained before. After the execution of the first three instructions, the AMT is given in Table I.

```assembly
movl $10, (%esp)
call malloc
movl %eax, -4(%ebp)
```

We keep track of the memory blocks which have been allocated and the corresponding pointers which are pointing to these memory blocks. Whenever a pointer is assigned to a new pointer, we keep track of this information using the forward slicing as explained.

Now for the next three instructions, which are the same,

```assembly
movl $10, -4(%ebp);
call malloc
movl %eax, -4(%ebp)
```

Here, let us assume the starting address to be 5000. Since the size of the block is 10 the upper-bound for this memory block will be 5009. After the execution of next three instructions, i.e., after the next allocation the table is given in Table II. Here a new memory block is allocated to the same
pointer making the first memory block inaccessible.

Here we use forward slicing to update the pointer field of the table. When we slice with respect to the pointer which is currently pointing to a memory block, we get all the pointers if any, to which this pointer has been assigned. At the end of execution we go over the AMT and if we find any memory block which is active and does not contain any references to it, we declare that we found a memory leak.

Case 2. When the last reference to a memory block is stored in a memory location which is freed, thereby loosing that reference and creating a memory leak. Consider the C code given in Fig. 4.

```c
void mem_leak3()
{
    node *n1;
    n1 = (node *)malloc(sizeof(node));
    n1->next = (node *)malloc(sizeof(node));
    free(n1);
    n1 = (node *)malloc(sizeof(node));
    return;
}
```

Fig. 4. Memory leak example C-code Case-2

The corresponding assembly code for this is given in Fig. 5.

```
movl $8, (%esp)
call malloc
movl %eax, -4(%ebp)
```

The allocated memory table after the first step is given in Table III.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Address</th>
<th>Size</th>
<th>No. Of Ref.</th>
<th>List of Ref.</th>
<th>Parent</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td>10</td>
<td>1</td>
<td>-4(%ebp)</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE I
ALLOCATED MEMORY TABLE FOR MEMORY LEAK, CASE 1, STEP 1

```
movl $8, (%esp)
call malloc
movl %eax, -8(%ebp)
movl -8(%ebp), %ebx
movl $8, (%esp)
call malloc
movl %eax, 4(%ebx)
movl -8(%ebp), %eax
movl %eax, (%esp)
call free
movl $8, (%esp)
call malloc
movl %eax, -8(%ebp)
addl $20, %esp
popl %ebx
popl %ebp
ret
```

Fig. 5. Assembly code of Fig. 4

We can observe in Table IV that the first allocated memory block contains a pointer to the newly allocated memory block. Hence the parent of the new memory block is the first memory block. Hence both the memory blocks are active at this point of time.

The last remaining step is as follows:

```
movl -8(%ebp), %eax
movl %eax, (%esp)
call free
```

In the instructions mentioned above, the first memory block is freed. We mark this block inactive now. At the end of execution we can the Allocated Memory Table, when we find a block with a parent we follow to that parent block and check if it’s active or not. If the parent block in inactive we declare the child block as a memory leak. Here also, we keep track of multiple pointers to the same memory block using forward slicing on the initial pointer. This yields us all the
other pointers pointing to this block, which gets updated in the AMT.

B. Buffer Overflows

A Buffer Overflow, or Buffer Overrun, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer. The result is that the extra data overwrites adjacent memory locations. The overwritten data may include other buffers, variables and program flow data, and may result in erratic program behavior, a memory access exception, program termination (a crash), incorrect results or - especially if deliberately caused by a malicious user - a possible breach of system security.

Buffer overflows can be triggered by inputs specifically designed to execute malicious code or to make the program operate in an unintended way. As such, buffer overflows cause many software vulnerabilities and form the basis of many exploits. Sufficient bounds checking by either the programmer, the compiler or the runtime can prevent buffer overflows.

1) Detection of Buffer Overflow: Here we concentrate on dynamically allocated buffers, since statically allocated buffers, if there are more than one in the current function, are present on the stack and do not have fixed bounds which are detectable in the assembly code. We use Hybrid Analysis for solving this case. In static part of the analysis forward slicing is used to keep track of pointers pointing to a memory block.

2) Methodology with Example:

1) When a malloc is called, by dynamic instrumentation using the PIN memory profiling tool we extract the start address of the memory block, it’s size and the address of the pointer in which this address is stored in.

2) This pointer is given an upper and lower bound depending on the size of the memory block.

3) When this pointer is assigned to a register which happens during a read or write access, the upper bound and lower bound values are passed onto the register.

4) When a write to a memory area occurs, as in

\[
\text{movl } 20, (\%eax)
\]

we extract the value of eax register and check if its bounds are present and if this is within those bounds.

5) The bounds are calculated using Backward Slicing as explained previously in III-B.

Let us consider the simple example in Fig. 6:

```c
void array_oob(void)
{
    char *a;
    char c;
    a = (char *)malloc(10);
    // array out of bound
    // reading and writing
    a[10] = ’a’;
    c = a[10];
    return;
}
```

Fig. 6. A Simple Example for Buffer Overflow

The corresponding assembly code for Fig. 6 is shown in Fig. 7.

The first 6 steps are as follows:

1 movl $10, (%esp)

2 call malloc

3 movl %eax, -8(%ebp)

4 movl -8(%ebp), %eax

5 addl $97, %eax

6 movb $1, (%eax)

The corresponding AMT for this code in Table VI. Assume that the start address of the allocated memory block is 5000.

\[
\text{Lower Bound for } -8(%\text{ebp}): 5000
\]
TABLE VI

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Address</th>
<th>Size</th>
<th>No. Of Ref.</th>
<th>List of Ref.</th>
<th>Parent</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td>10</td>
<td>1</td>
<td>-8(%ebp)</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Allocated Memory Table For Buffer Overflow

```
array_oob:
  ...
  subl $24, %esp
  movl $10, (%esp)
  call malloc
  movl %eax, -8(%ebp)
  addl $10, %eax
  movb $97, (%eax)
  addl $10, %eax
  movzbl (%eax), %eax
  movb %al, -1(%ebp)
  leave
  ret
```

Fig. 7. Assembly code for Fig. 6

**Upper Bound for -8(%ebp): 5009**

In the step-6 `movb $1, (%eax)`, we extract the value of the eax register. When we backward slice from this step on eax, we can see that eax was most recently defined in step-4. Here the bounds on -8(%ebp) are passed onto the register eax.

Now from the AMT we get the upper bounds and lower bounds on this particular register as 5000 and 5009 respectively. Here instructions like addl, subl etc., do not affect the bounds already present on a register since this register is being used here. Only movl like instructions will affect the bounds of the register. If we find out that the value of eax is out of these bounds, then we can say that a buffer overflow has occurred.

**C. Dangling Pointers**

Consider the code given in Fig. 8. Equivalent assembly code (after instrumentation) is given in Fig. 9. We observe the following in the assembly code given in Fig. 9.

```
int *call_func(void);
int main()
{
  int *p, *q, *r;
  p = (int *)malloc(2);
  q = p;
  *p = 2;
  *q = 3;
  free(p); // Memory pointed to by p is freed.
  r = (int *)malloc(2);
  *r = 500;
  *q = 10; // q is the dangling pointer here since it is pointing to the location which is already freed.
  printf("\n The value of r = \"d\n", *r);
}
```

Fig. 8. Example: Dangling Pointers

```
main:
  1: movl %ebp , templ
  2: movl %esp, %ebp
  3: subl $24, %esp
  4: andl $-16, %esp
  5: movl %0, %eax
  6: addl $15, %eax
  7: addl $15, %eax
  8: shr $4, %eax
  9: sall $4, %eax
 10: subl %eax, %esp
 11: movl %2, (%esp)
 12: call malloc
 13: movl temp10 , %eax #instrumented
 14: movl %eax, -4(%ebp)
 15: movl -4(%ebp), %eax
 16: movl %eax, -8(%ebp)
 17: movl -8(%ebp), %eax
 18: movl $2, (%eax)
 19: movl -8(%ebp), %eax
 20: movl $3, (%eax)
 21: movl -4(%ebp), %eax
 22: movl %eax, (%esp)
 23: call free
 24: movl $2, (%esp)
 25: call malloc
 26: movl temp10 , %eax #instrumented
 27: movl %eax, -12(%ebp)
 28: movl -12(%ebp), %eax
 29: movl $500, (%eax)
 30: movl -8(%ebp), %eax
 31: movl %10, (%eax)
 32: movl -12(%ebp), %eax
 33: movl %10, (%eax)
 34: movl %eax, 4(%esp)
 35: movl $.LC0, (%esp)
 36: call printf
 37: leave
 38: ret
```

Fig. 9. Assembly Code of Fig. 8

1) -4(%ebp) is the location where the content of p (refer C code) is saved (line 14).
2) -8(%ebp) is the location where the content of q (refer C code) is saved (line 16).
3) Memory pointed to by p is freed on line 23.
4) On line 30, q is again accessed, after the location it pointed to was already freed (line 23), hence it is a dangling pointer.

1) Instrumentation: We perform the following transformations in the code to enhance analysis.
1. Make the code stackless

pushl ebp → movl ebp , temp1
This is done to make the code stackless and slicing easier. Note that this equivalent code gives the same result with respected to used and defined variables. ebp is the used variable in both the cases:

2. Inclusion of instruction 13 and 26

13: movl temp10 , %eax
26: movl temp10 , %eax

The pointer returned after calling malloc is stored in eax. To make it explicitly visible in the code. Instruction 13 and 26 are added. temp10 is a temporary variable. Here eax is the defined variable to remove implicit assigning of eax

Steps for Detection Algorithm are as follows:-

1) Backward slice with respect to the pointer(say p) which is used in freeing the memory. This gives us all the instructions which can effect the value of the pointer. (Here instruction 22 is used to perform backward slicing. List of instructions obtained are as follows: 22, 21, 14, 13, 10, 9, 4).

2) For each instruction obtained above, perform forward slicing to get the list of instructions.

3) Take the union of all the sets of instructions obtained in step 2 and store them in a cumulative slice array.

4) If there exists an instruction in cumulative slice array which does not contain esp and comes after the instruction call free is executed, then report the presence of dangling pointers in the code.

2) Example Results: Forward slices of each instruction in step 2 in our example are shown in Table VII :-

<table>
<thead>
<tr>
<th>Forward Slice start node</th>
<th>List of nodes obtained in the slice</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>21, 22</td>
</tr>
<tr>
<td>14</td>
<td>14, 21, 22, 17, 18, 15, 16, 30, 31, 19, 20</td>
</tr>
<tr>
<td>13</td>
<td>13, 14, 21, 22, 17, 18, 15, 16, 30, 31, 19, 20</td>
</tr>
<tr>
<td>10</td>
<td>10, 35, 34, 24, 22, 11</td>
</tr>
<tr>
<td>9</td>
<td>9, 10, 35, 34, 24, 22, 11</td>
</tr>
<tr>
<td>4</td>
<td>4, 10, 35, 34, 24, 22, 11</td>
</tr>
</tbody>
</table>

CUMULATIVE SLICES OBTAINED IN STEP 2 (DANGLING POINTERS)

- Cumulative slice obtained in Step 3 is as follows:- 22, 21, 14, 17, 18, 15, 16, 30, 31, 19, 20, 13, 10, 35, 34, 24, 11, 9 and 4.
- List of instructions that qualify the condition in Step 4 are 30, 31.

The results obtained in Step 4 show the presence of dangling pointer.

V. CONCLUSIONS

Since the source code is not usually available, analysis on the executable has gained importance to ensure trustworthiness in the system. It can be seen that static or dynamic analysis alone is inefficient and insufficient to extract security vulnerabilities. Hence, in this a hybrid approach is taken to detect security vulnerabilities like memory leaks, buffer overflows and dangling pointers. In dangling pointers, the code is made stackless by adding new instructions to retain the semantics of the code in order to enhance the detection analysis. PDG is constructed to get the slice with respect to the slice criterion, so that the analysis is more focussed. PIN tool is used to extract the control flow and register bounds which are used in static analysis to detect the memory errors mentioned above. This approach can also be extended to analyzing components of the system.

REFERENCES