Hybrid Analysis Of Executables To Detect Security Vulnerabilities

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

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

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

ABSTRACT

Detection of vulnerabilities in executables is one of the major challenges facing the software industry and is mainly due to the unavailability of the source code. In this work, we present a hybrid approach which is a combination of static and dynamic analysis to identify vulnerabilities. In this approach, we first instrument the executable using PIN [3] to extract the control flow and the corresponding assembly code using disassembler. We then perform static analysis on the assembly code for constraint bound checking using control flow and register bounds. In this way, we exploit the synergy between static and dynamic analysis to detect memory leaks, buffer overflow and dangling pointers.

Categories and Subject Descriptors

K.6.5 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Security and Protection—*Invasive software, Unauthorized access*

General Terms

Algorithm, Design, Experimentation, Security.

Keywords

Hybrid analysis, security vulnerabilities, memory errors, instrumentation, slicing.

1. INTRODUCTION

In the interconnected world of computers, the malicious code has become an omnipresent and dangerous threat which can enter hosts using variety of ways such as exploiting the software flaws, hidden functionality in regular programs and social engineering. The state of the art program analysis tools work at source code level. Since most of the commercial softwares specially on windows platform and malicious software such as Virus, Trojans, Spyware are distributed in the form of binary code, it becomes very important to analyze binary codes.

Errors introduced by programmers can also be exploited by malicious code writers. For example, low level programming languages (e.g., C and C++) provide flexibility for programmers to access arbitrary memory accesses within the program's address space in the form of pointers. When the programmers are not careful enough using these features, it might result in illegal memory reads/writes. These illegal reads/writes are then exploited by malicious code writers to tamper with the system. Buffer overflow errors are another highly exploited errors to introduce maliciousness in the code. These errors can be used for stack smashing.

In the absence of source code, various static and dynamic analysis techniques on the binary code are adapted to detect security vulnerabilities. Both the 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.

2. METHODOLOGY

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. We also extract information about the dynamic memory blocks allocated during runtime. In the second step, we instrument the code to make it analyzable. In the third step, we perform static slicing [1] 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 [2] to get the more precise slice. Also we build up tables using dynamic analysis as we explain below and decide if there are errors in the executables depending on the contents of the table.

For detection of memory access errors, 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 and 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 contains a pointer 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). Memory access errors are typically classified into three types, which are

ISEC'09, February 23–26, 2009, Pune, India.
ACM 978-1-60558-426-3/09/02.

S.No.	Addr	Size	#Refs.	List Ref	Parent	Active
1	5000	8	1	-	-	0
2	5008	8	1	4(%ebx)	1	1

Table 1: Allocated Memory Table for memory leak

memory leaks, array index out of bounds (buffer overflows) and dangling pointers.

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

Figure 1: A memory leak example in C-code

2.1 Detection of memory leaks :: Consider the C code given in Fig. 1. The corresponding assembly code for this is given in Fig. 2.

Whenever a pointer is assigned to a new pointer, we keep track of this information using the forward slicing [1] After the first ten assembly instructions, the allocated memory table (AMT) is given in Table 1. The first memory block is freed and is marked inactive. At the end of execution we scan the allocated memory table. When we find a block with a parent we look up the entry of the parent block in the table and check if it's active or not. If the parent block is marked 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.

2.2 Detection of buffer overflow :: Here we concentrate on dynamically allocated buffers, since the bounds of individual buffers are not detectable in case of multiple statically allocated buffers. We use hybrid analysis to solve this case. In static part of the analysis forward slicing is used to keep track of pointers pointing to a memory block. The methodology for detecting buffer overflow is as follows:–

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

```
mem_leak3:
        ..
        subl    $20, %esp
        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
        ..
```

Figure 2: Assembly of the code in Fig. 1

2. This pointer is given an upper and lower bound based 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 register inherits the upper and lower bounds of the pointer.

4. When a write to a memory area occurs, as in: *movl* $20, (%eax), we extract the address value in the *eax* register and check if the register has any defined bounds. If it has defined bounds then we check if the extracted address is within the bounds. If bounds are not defined then they are calculated using Backward Slicing [1] on the register. We then check if the extracted address is within the bounds.

We use the AMT similar to the one used for memory leaks to detect buffer overflow.

2.3 Dangling pointers :: We perform the following transformations in the code to enhance the analysis – The return value after a call instruction is stored in *%eax*. To make this value explicitly visible in the code we make the code stackless by moving this value to a temporary variable. Also, whenever there is any stack operation like push or pop, we use a temporary variable. The detection algorithm is as follows:–

1. Backward slicing is done with respect to the pointer which is used in freeing the memory to get all the instructions and variables which can effect the value of the pointer.

2. For each instruction listed in step 1 forward slicing is performed to get cumulative slice.

3. If there exists an instruction in cumulative slice which does not contain *esp* and comes after the instruction *call free* is executed, the presence of dangling pointer in the code is reported.

In this brief note, we have demonstrated detection of a few typical types of unauthorized memory accesses from *elf* object files using hybrid approach, which can be extended to other binary formats as well. Moreover, the approach can be used to significantly reduce the size of control flow graphs (CFG) where indirect function calls are extensively used. The executable can be run on representative inputs in a controlled environment to obtain the possible run-time address of each statically unresolved indirect call site after the CFG is completed. The work can be extended to various other issues in security vulnerabilities.

3. REFERENCES

[1] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises, pages 184–189, 1999.

[2] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *In WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[3] C. Keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200. ACM Press, 2005.