International Journal of Computer Applications (0975 – 8887)
Volume 104 – No.6, October 2014

Compiler for Detection of Program Vulnerabilities

Abhishek Nayyar
Department of Computer Science, NIT Jalandhar
NIT Jalandhar

Arun Kumar
Department of Computer Science, NIT Jalandhar
NIT Jalandhar

Umang Saxena
Department of Electronics and communication, NIT Allahabad
NIT Allahabad

ABSTRACT

Program Vulnerabilities may be unwarranted for any organization and may lead to severe system failure. Due to the advancement of technology there has been increase in the area of vulnerability attacks which are exploited by hackers for getting access to the system or insertion of their malicious code. In this paper we present a proposal for compiler design which prevents some common vulnerability. The output result for our compiler would be compile time warning stating the possible vulnerability in the code. We will also look into the details about the different type of vulnerability and how the attacker can exploit those vulnerabilities in order to corrupt the system. The knowledge of various vulnerability creation areas have been used to design a compiler for vulnerability prevention. Compiler in this publication uses the symbol table generation mechanism for syntactically, semantically segregation of executable code and canary guard mechanism for the protection of cases of buffer overflow. Major work in this area deals with the simple scenarios for vulnerability detection but our aim is to check for various complicated scenarios and non common possibilities for program attack and designing a framework preventing such kinds of attacks.

General Terms

Lexical Analysis, Syntax Analysis, Parser, Token, Semantic Analyzer, Symbol table, Random XOR.

Keywords

Program vulnerabilities, Stack smashing, Buffer overflow, Canary guard, Compiler, Canaries, Terminator.

1. INTRODUCTION

Simple program vulnerabilities can cause severe damage to even the most sophisticated and well constructed systems causing huge loss of finances resources, consumer privacy, data, etc. Exposing and identifying security vulnerabilities is notoriously difficult; research efforts in software testing focus almost exclusively on common case; i.e., the program behavior that users are likely to encounter when they use the program correctly. This approach is not conducive to exposing security flaws as vulnerabilities are typically found using inputs that users would not normally enter. Consider the typical stack smashing attack which seeks to overflow a program buffer and trick the program into running arbitrary code. Such an attack would require the user to enter the binary code for particular instructions which is improbable at best.

The lack of testing strategies targeted towards security concerns results in the software community being more reactive than proactive with respect to security vulnerabilities. Software Engineers currently have no easy of testing for security problems, thus problems are typically found after the software has been released. Once a program’s vulnerabilities have been discovered, programmers typically, modify the code to add a security mechanism tailored to the known vulnerability and the program. The best solution would be to engineer programs so that vulnerabilities are not present, but this is not entirely possible, primarily because attackers continue to find new vulnerabilities. A variety of strategies for preventing vulnerabilities have been proposed involving all aspects of the program and its execution environment. These techniques can be broadly termed program-based as they focus upon the program or its execution environment.

Testing of these techniques is often poor. A program with a known vulnerability is found and recompiled with a particular protection scheme. The particular input that exploited the vulnerability is then provided to the program to determine whether the protection scheme succeeded. This testing scheme does not inspire great confidence in the security mechanism since it could only be tried on a few programs with one particular input triggering a particular type of vulnerability. In this paper, we present the design of a framework which enables the automatic and systematic testing of various security mechanisms. The key insight is that such mechanisms can be tested without resorting to specially designed test cases using dynamic compilation technology. Dynamic compilers are particularly well-suited for this problem because they can enable a user to modify program state and instructions during the execution of the program. The broad impact of this framework will be increased confidence in security mechanisms developed for program-based vulnerabilities as well as a framework for experimental investigation into new security mechanisms.

2. BACKGROUND

2.1 Attack History

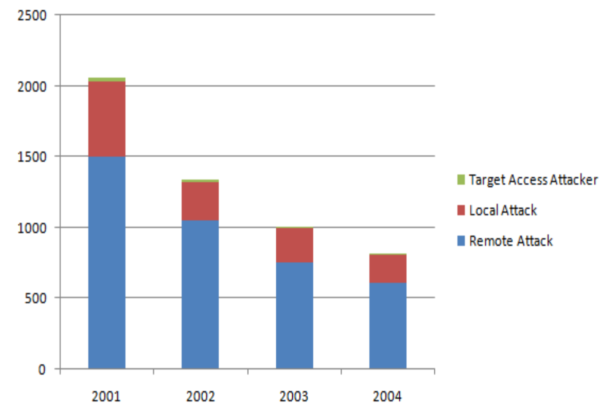
The July 2005 announcement by computer security researcher Michael Lynn at the Black Hat security conference of a software flaw in Cisco Systems routers grabbed media attention worldwide. The flaw was an instance of a buffer overflow; a security vulnerability that has been discussed for 40 years yet remains one of the most frequently reported types of remote attack against computer systems. In 2004, the national cyber-security vulnerability database (nvd.nist.gov) reported 323 buffer overflow vulnerabilities, an average of more than 27 new instances per month. For the first six months of 2005, it reported 331 buffer overflow vulnerabilities. Meanwhile, securities researchers have sought to develop techniques to prevent or detect the exploitation of these vulnerabilities. Here, we discuss what buffer overflow attacks are and survey the techniques that can be used to mitigate their threat to computer systems [5].

Table 1. Data summarization of major attacks affecting systems in past

Attack Source	Date of Attack	Attack type	Affect
Comair Airline	Dec 25,2005	Integer overflow	1100 flights were grounded
Unix OS	Feared Jan 19,2038	Integer overflow	Income tools
Morris Worm	1998	Buffer overflow	Internet shut down
AOL 's AIM	2004	Buffer overflow	Attack possibility on user click
Blaster Worm	August,2003	Buffer overflow	Corrupted Microsoft window system
IE 4.0 & 4.1	Nov 12,1997	Buffer overflow	Affected IE behavior

Malicious code is any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function. Although the problem of malicious code has a long history, a number of recent, widely publicized attacks and certain economic trends suggest that malicious code is rapidly becoming a critical problem for industry, government, and individuals. Attack scripts are programs written by experts that exploit security weaknesses, usually across the network, to carry out an attack.

- Attack scripts exploiting buffer overflows by "smashing the stack" are the most commonly encountered variety.
- Java attack applets are programs embedded in Web pages that achieve foothold through a Web browser.
- Dangerous ActiveX controls are program components that allow a malicious code fragment to control applications or the operating system.



Graph 1. Attack summarization on the basis of year [5]

2.2 Early developments to the problem

There has been development of static source analysis technique for vulnerability detection in C based on the combination of taint analysis and value range propagation technique used for compiler optimization [1]. There have been work specific to cross 86 platform for virtual execution environment that combines information from compositional, static, and dynamic program analysis to identify vulnerabilities and timing channels, and uses code transformations to prevent those from being exploited [2]. RICH (Run time integer Checker) for detection of integer based attacks in C [3]. The approach used by the above methods is related to specific type of attack. We present the solution of different types of attacks in our design of compiler for vulnerability detection.

3. ATTACK DETECTION

Before moving into the concept of compiler design for the attack prevention it's important to understand the various methods of detecting any attack. For understanding the method of attack detection, we must have some insight about the types of attacks. This section presents the in depth description and analysis of the various program-based attacks implemented in this project. The various attacks which have been implemented here are stack smashing, buffer overflow, declaration attack.

3.1 Stack Smashing

In software, a stack smashing (also known as stack buffer overflow) occurs when a program writes to a memory address on the program's call stack outside of the intended data structure; usually a fixed length buffer. Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than there was actually allocated for that buffer. This almost always results in corruption of adjacent data on the stack, and in cases where the overflow was triggered by mistake, will often cause the program to crash or operate in undesirable way. This type of overflow is part of the more general class of programming bugs known as buffer overflows.

Exploiting stack overflow: The canonical method for exploiting a stack based buffer overflow is to overwrite the function return address with a pointer to attacker-controlled data (usually on the stack itself).This is illustrated in the example below:

```
void fun(char *s)
{
        char buffer[4];
        strcpy(buffer,s);
        printf(" value in buffer %s \n",buffer);
}
void extra()
{
        printf(" Function is case study of stack smashing \n");
}
void main(int argc,char *argv[])
{
        fun(argv[1]);
}
```

This code takes an argument from the command line and copies it to a local stack variable c. This works fine for command line arguments smaller than 4 characters. Any arguments larger than 4 characters long will result in corruption of the stack. (The maximum number of characters that is safe is one less than the size of the buffer here because in the C programming language strings are terminated by a zero byte character. A four-character input thus requires five bytes to store; the input followed by the sentinel zero byte. The zero byte then ends up overwriting a memory location that's one byte beyond the end of the buffer.)

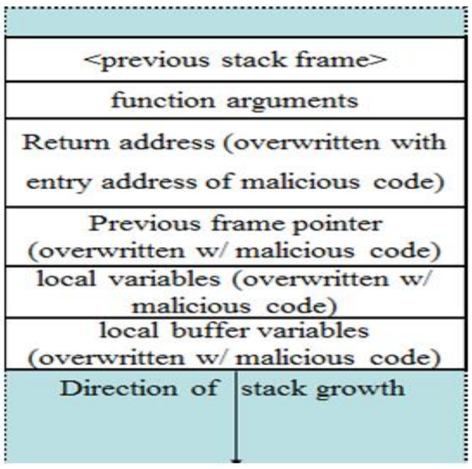
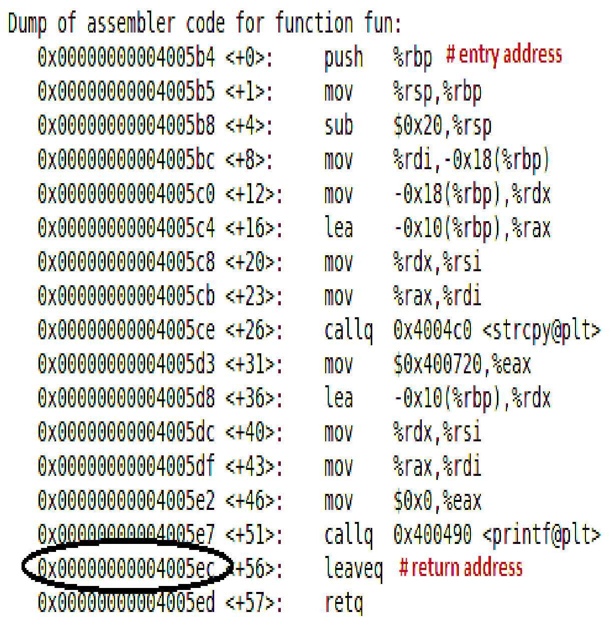

<previous stack frame>
function arguments
Return address (overwritten with entry address of malicious code)
Previous frame pointer (overwritten w/ malicious code)
local variables (overwritten w/ malicious code)
local buffer variables (overwritten w/ malicious code)
Direction of stack growth

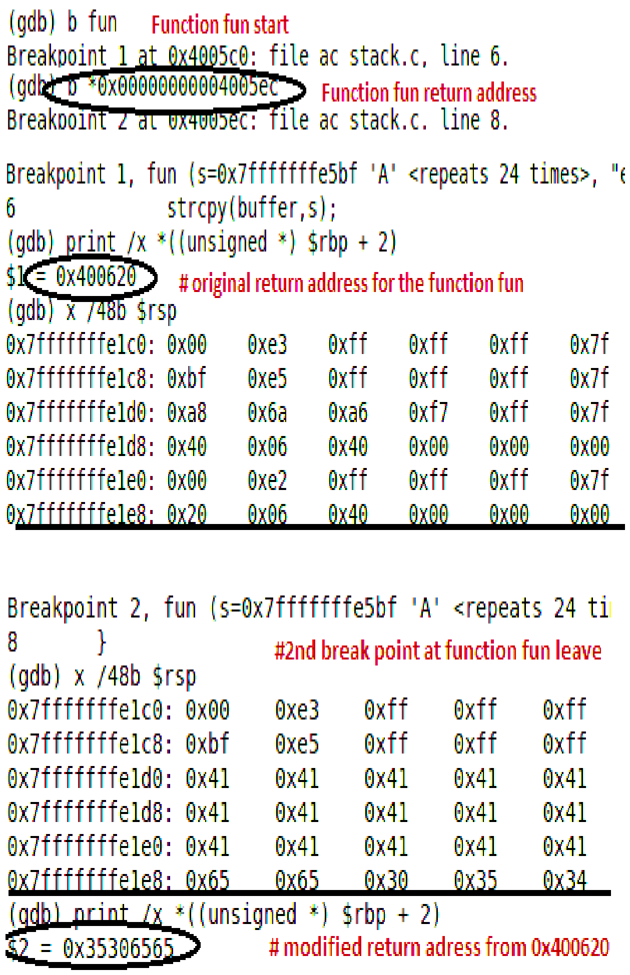
Fig 1. Data storage representation in stack



```
Dump of assembler code for function fun:
   0x00000000004005b4 <+0>:    push   %rbp  #entry address
   0x00000000004005b5 <+1>:    mov    %rsp,%rbp
   0x00000000004005b8 <+4>:    sub    $0x20,%rsp
   0x00000000004005bc <+8>:    mov    %rdi,-0x18(%rbp)
   0x00000000004005c0 <+12>:   mov    -0x18(%rbp),%rdx
   0x00000000004005c4 <+16>:   lea    -0x10(%rbp),%rax
   0x00000000004005c8 <+20>:   mov    %rdx,%rsi
   0x00000000004005cb <+23>:   mov    %rax,%rdi
   0x00000000004005ce <+26>:   callq  0x4004c0 <strcpy@plt>
   0x00000000004005d3 <+31>:   mov    $0x400720,%eax
   0x00000000004005d8 <+36>:   lea    -0x10(%rbp),%rdx
   0x00000000004005dc <+40>:   mov    %rdx,%rsi
   0x00000000004005df <+43>:   mov    %rax,%rdi
   0x00000000004005e2 <+46>:   mov    $0x0,%eax
   0x00000000004005e7 <+51>:   callq  0x400490 <printf@plt>
   0x00000000004005ec <+56>:   leaveq #return address
   0x00000000004005ed <+57>:   retq
```

Fig 2. Assembler code for function fun in above program

The above figure shows the entry point and the return point values for the function fun. Give application a very long string with malicious code. The string length, being much larger than the space allocated in the heap (buffer size declaration) causes the heap to overflow into the stack and overwrites the return address. The return address now points to the beginning of the malicious code. We can add the extra character which is more than our buffer size to be our entry address off another function. Following is depiction for stack smashing case when length of string is greater than the length of the buffer.



```
(gdb) b fun   Function fun start
Breakpoint 1 at 0x4005c0: file ac stack.c, line 6.
(gdb) b *0x00000000004005ec   Function fun return address
Breakpoint 2 at 0x4005ec: file ac stack.c, line 8.

Breakpoint 1, fun (s=0x7fffffffe5bf 'A' <repeats 24 times>, "
6               strcpy(buffer,s);
(gdb) print /x *((unsigned *) $rbp + 2)
$1 = 0x400620   # original return address for the function fun
(gdb) x /48b $rsp
0x7fffffffe1c0: 0x00    0xe3    0xff    0xff    0xff    0x7f
0x7fffffffe1c8: 0xbf    0xe5    0xff    0xff    0xff    0x7f
0x7fffffffe1d0: 0xa8    0x6a    0xa6    0xf7    0xff    0x7f
0x7fffffffe1d8: 0x40    0x06    0x40    0x00    0x00    0x00
0x7fffffffe1e0: 0x00    0xe2    0xff    0xff    0xff    0x7f
0x7fffffffe1e8: 0x20    0x06    0x40    0x00    0x00    0x00


Breakpoint 2, fun (s=0x7fffffffe5bf 'A' <repeats 24 ti
8       }
                              #2nd break point at function fun leave
(gdb) x /48b $rsp
0x7fffffffe1c0: 0x00    0xe3    0xff    0xff    0xff
0x7fffffffe1c8: 0xbf    0xe5    0xff    0xff    0xff
0x7fffffffe1d0: 0x41    0x41    0x41    0x41    0x41
0x7fffffffe1d8: 0x41    0x41    0x41    0x41    0x41
0x7fffffffe1e0: 0x41    0x41    0x41    0x41    0x41
0x7fffffffe1e8: 0x65    0x65    0x30    0x35    0x34
(gdb) print /x *((unsigned *) $rbp + 2)
$2 = 0x35306565         # modified return adress from 0x400620
```

Fig 3. Breakpoint at entry and exit along with the stack data after the hit of first break point

The following figures show the return address of function getting modified when entered string has size greater than the size of buffer. This is termed as stack smashing.

3.2 Buffer Overflow

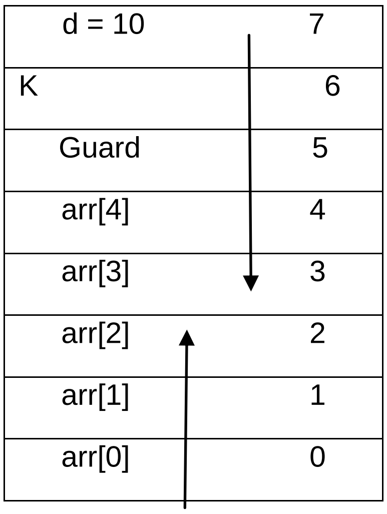
Buffer Overflows is one of the most common vulnerabilities in software. It is particularly problematic when present in system libraries and other code that runs with high execution privileges [6]. When we normally allocate some buffer of fixed size and in place of providing the data in accordance with the size of the buffer, we provide some additional data. In that case the additional data may overwrite some useful information in the consecutive locations, which may lead to various system issues. In this section we will look at a simple case for buffer overflow in which providing data of size more than its allocated memory may lead to modification of other values stored at the adjacent locations in stack.

Exploiting buffer overflow: The canonical method for exploiting a buffer overflow is to overwrite the values at adjacent locations with a pointer to attacker-controlled data (usually on the stack itself).This is illustrated in the example below:

```
main()
{
        int d=10;
        int k;
        printf("%d\n",d);
        int arr[5]={7,9,3,12,8};
        arr[6]=30;
        printf("%d\n",d);
        arr[3]=k;
        printf(" value k %d\n",k);
}
```

The code has the reserved memory location of 5 bytes for array but it is trying to overwrite the value at sixth location with the new value. Sixth location corresponds to the value of local variable k , whereas the seventh location corresponds to the local variable d. Overwriting the value at location six causes the modification in the value of variable six which is stored at this address. Let us look at the detail analysis of such type of vulnerability.

Table 2. Stack depiction during buffer overflow



d = 10	7
K	6
Guard	5
arr[4]	4
arr[3]	3
arr[2]	2
arr[1]	1
arr[0]	0

As we can see in the above figure any modification in the address location corresponding to the arr[6] will cause modification in the value of k. The following table illustrates the buffer overflow attack with the modification in the value of undesired variable.

Table 3. Value corresponding to various address during buffer overflow

Variable	Address	Value
d	0x7fffffffe20c	0x0000000a
k	0x7fffffffe208	Dummy
Arr[0]	0x7fffffffe1f0	0x00000007
Arr[1]	0x7fffffffe1f4	0x00000009
Arr[2]	0x7fffffffe1f8	0x00000003
Arr[3]	0x7fffffffe1fc	0x0000000c
Arr[4]	0x7fffffffe200	0x00000008
Arr[5]	0x7fffffffe204	0x00007fff (Guard)
Arr[6]	0x7fffffffe208	0x0000001e (**modified k**)
Arr[7]	0x7fffffffe20c	**(can modify d)**

3.3 Declaration attack

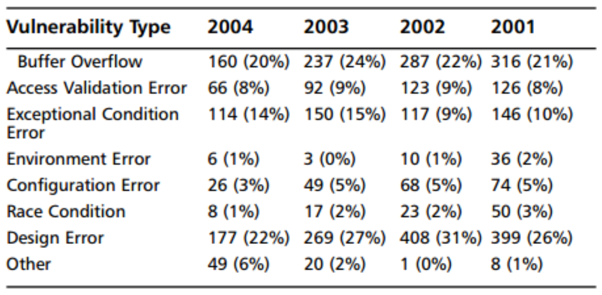
In programming languages, a **declaration** specifies the identifier, type, and other aspects of language elements such as variables and functions. It is used to announce the existence of the element to the compiler; this is important in many strongly typed languages (such as C) that require variables and their types to be specified with a declaration before use, and is used in forward declaration. Before assigning any variable A a value of variable B, variable B must be properly defined. When we don't assign any value to the variable and intern assigns that same variable to another variable, it is possibility that both variables may end up containing any garbage value which is also categorized as a type of vulnerability for the system.

Exploiting declaration attacks: The canonical method for exploiting a declaration attack is to assign any undefined variable to some new variable. This will cause both the variables to have some garbage value .This is illustrated in the example below:

```
func()
{
        int x;
        int y = x;
}
```

Here both x and y will contain some dummy values, which may lead to system corruption in future. So it's important to detect such type of attack before only.

Fig 4. Vulnerability summary on the basis of type of vulnerability



Vulnerability Type	2004	2003	2002	2001
Buffer Overflow	160 (20%)	237 (24%)	287 (22%)	316 (21%)
Access Validation Error	66 (8%)	92 (9%)	123 (9%)	126 (8%)
Exceptional Condition Error	114 (14%)	150 (15%)	117 (9%)	146 (10%)
Environment Error	6 (1%)	3 (0%)	10 (1%)	36 (2%)
Configuration Error	26 (3%)	49 (5%)	68 (5%)	74 (5%)
Race Condition	8 (1%)	17 (2%)	23 (2%)	50 (3%)
Design Error	177 (22%)	269 (27%)	408 (31%)	399 (26%)
Other	49 (6%)	20 (2%)	1 (0%)	8 (1%)

4. ATTACK PREVENTION

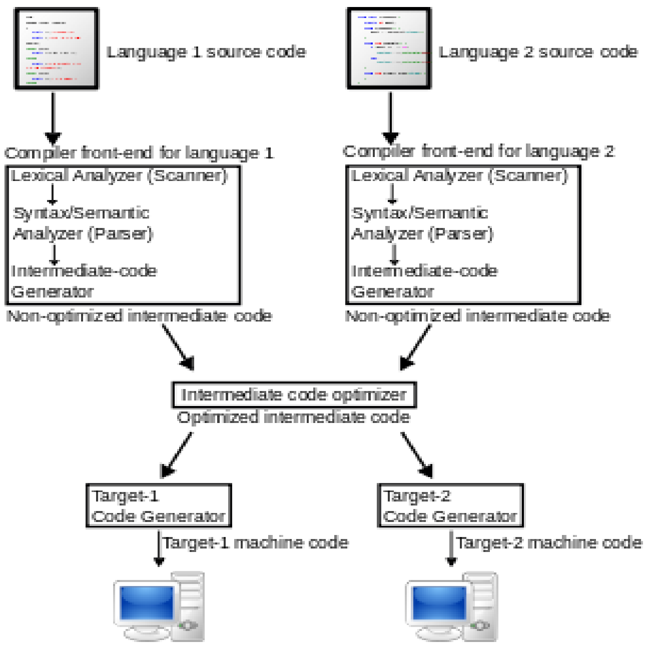
4.1 Compiler design

The main target of this publication is to design a compiler which warns the user of various vulnerabilities. The vulnerability warnings are included in the list of compile time warnings. Before understanding the detailed mechanism of programming vulnerabilities detection, we will look into the design of compiler phases and some basic terminology associated with compiler.

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program [7].

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language. In this project, we deal with a typical 'language translator'. A compiler is likely to perform many or all of the following operations: lexical analysis, pre processing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization. For sake of detection of vulnerabilities, we, hereby, exclude from our discussion phases of code generation and code optimization [8].

Fig 5. Phases of compiler [4]


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

Processing during compiler phases

A. A high-level program that takes as input another high-level program as a string and splits into the desired program.

B. Inserting of instructions for attack into the program like stack smashing, overflow etc.

C. Modifying of the return address; changing of the flow of execution of instructions in the program.

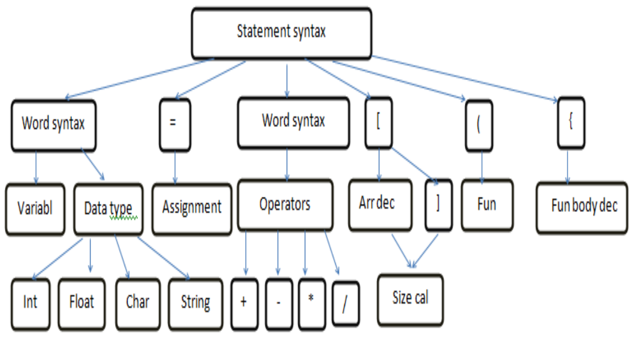
4.1.1 Lexical Analysis

Lexical analysis is the process of analyzing a stream of individual characters (normally arranged as lines), into a sequence of lexical tokens (tokenization. for instance of "words" and punctuation symbols that make up source code) to feed into the parser. Roughly it is equivalent to splitting ordinary text written in a natural language (e.g. English) into a sequence of words and punctuation symbols. In lexical phase of compiler each word is categorized as a token. A token is a categorized block of text, usually consisting of indivisible characters known as lexemes. A lexical analyzer initially reads in lexemes and categorizes them according to function, giving them meaning. This assignment of meaning is known as tokenization.

4.1.2 Syntax Analysis

This is alternatively known as parsing. It is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) is grammatically correct (without worrying about meaning).The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Note that this sequence need not be meaningful; as far as syntax goes, a phrase such as "**true** + 3" is valid but it doesn't make any sense in most programming languages. The parser takes the tokens produced during the lexical analysis stage, and attempts to build some kind of in-memory structure to represent that input. Frequently, that structure is an 'abstract syntax tree' (AST).

Fig 6. AST (Abstract syntax tree) representation of code for compiler execution.



4.1.3 Semantic Analyzer

This phase of compiler design deals with analyzing the execution code by categorization into symbol table. A symbol table is a major data structure used in a compiler:

- Associates attributes with identifiers used in a program
- For instance, a type attribute is usually associated with each identifier
- A symbol table is a necessary component
- Definition (declaration) of identifiers appears once in a program
- Use of identifiers may appear in many places of the program text
- Identifiers and attributes are entered by the analysis phases
- When processing a definition (declaration) of an identifier

- In simple languages with only global variables and implicit declarations.
- The scanner can enter an identifier into a symbol table if it is not already there
- In block-structured languages with scopes and explicit declarations:
- The parser and/or semantic analyzer enter identifiers and corresponding attributes
- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared).

Table 4. Type table

Type	Value
Int	0
Float	1
Double	2
Char	3

Table 5. Parsing table

No	Variable	Type	Defined	Address
1	a	0	0	xx
2	b	3	1	xx
3	c	2	0	xx
4	d	1	1	xx

After parsing the code, compiler fills the data according to the type and parsing table. This information is used by compiler at the time of attack prevention.

4.2 Prevention of vulnerability cases

After getting the insight of various vulnerability attacks and compiler design, our next task is to understand the technique deployed by compiler in preventing such attacks.

4.2.1 Stack smashing and Buffer overflow

The basic problem of stack smashing has its root cause as buffer overflow. As the size of data increases the desired space than the possibility that it modifies the return address becomes immense. The extra value can be linked with the entry address of malicious code. So we can say our root cause of various vulnerabilities is buffer overflow. Our vulnerability prevention technique mainly focuses on preventing the root cause which is buffer overflow. Buffer overflow protection refers to various techniques used during software development to enhance the security of executable programs by detecting buffer overflows on stack-allocated variables as soon after they occur as is practical, and preventing them from becoming serious security vulnerabilities.

Table 6. Function return address table

Function Name	Return Address
Fun1	F1xx
Fun2	F2xx
Fun3	F3xx

Table 7. Function execution address

Function 1 return address F1xx changes to F1xy
Var1
Var2
Var3
.
.

During the return of the program, checked that Fun1 address should match with Fun1 (table 6 value).

If not matched then there is the stack smashing, and return the function to the actual return address that store in the function table If matched with the table value then return. Main mechanism of buffer overflow prevention is based on **canaries.**

Canaries

Canaries are pre known values which are placed between a buffer and control data on the stack in order to check the possibility of buffer overflows. In the case of buffer overflows, the first data to be corrupted will be canary which will result in the failure of verification of the canary value. This results in the invalidation of data and compiler warning for the case of vulnerability detection. This concept is based on the references of the historic practice of using canaries in field of coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a warning system. The use of canary in order to prevent buffer overflow is based on three method of its use. These are **Terminator, Random, and Random XOR canaries**.

Terminator canaries

Terminator Canaries use the observation that most buffer overflow attacks are based on certain string operations which end at terminators. The extra space for any string is reserved for its terminator. We can analyze the canary character in a string. If the value of the canary character is modified then it is the possibility for buffer or stack smashing case.

Random canaries

Random canaries can be generated randomly. This is achieved usually from an entropy-gathering daemon, so as to prevent an attacker from knowing their value. Usually, it is not logically possible to read the canary for exploiting; the canary is a secure value known only by those who need to know it—the buffer overflow protection code in this case.

Normally, a random canary is generated at program initialization, and stored in a global variable. This variable is protected by padding of unmapped pages, in order to avoid any attempt to read it using any kinds of tricks which may lead to some kind of segmentation fault or some abnormal termination in ram. If the attacker knows the canary location, it may still be possible to read the canary.

Random XOR canaries

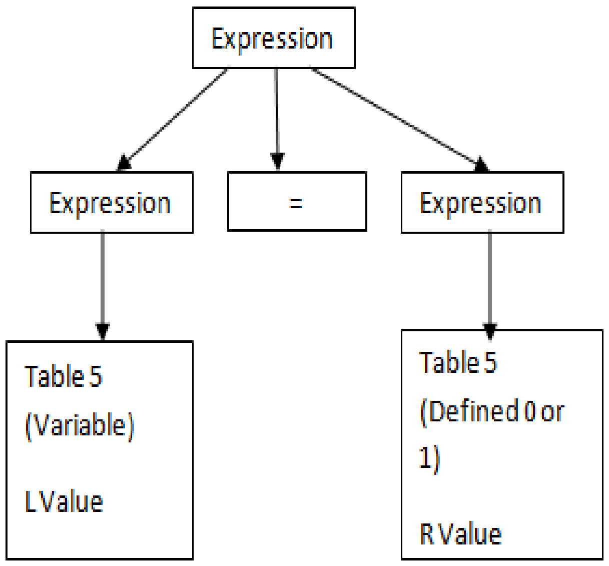
Random XOR Canaries are Random Canaries that are XOR scrambled using all or part of the control data. In this way, once the canary or the control data is clobbered, the canary value is wrong. Random XOR Canaries have the same vulnerabilities as Random Canaries, except that the 'read from stack' method of getting the canary is a bit more complicated. The attacker must get the canary, the algorithm, and the control data to generate the original canary for re-encoding into the canary he needs to use to spoof the protection.

In addition, Random XOR Canaries can protect against a certain type of attack involving overflowing a buffer in a structure into a pointer to change the pointer to point at a piece of control data. Because of the XOR encoding, the canary will be wrong if the control data or return value is changed. Because of the pointer, the control data or return value can be changed without overflowing over the canary.

4.2.2 Declaration attacks

After parsing of executable code we update the value corresponding to the defined column (table 5). At the start of the executable stage for any program code we look into the value of declared variables in parsing table. If any variable in parsing table is not defined, warning would be raised as not defined variable. Parsing of definition check is based on the flow diagram in Fig 7.

Fig 7. Flow diagram for definition check



5. CONCLUSION

We have presented the design of a framework allowing the testing of security frameworks and detection of program vulnerabilities on the basis of program-based attacks. Such a framework would allow for more efficient testing of these mechanisms, without resorting to complex methodologies. The key insight of this framework is that dynamic compilation technology allows us to insert and simulate attacks during program execution.

6. FUTURE SCOPE

Various areas outlined briefly in this section are open research issues that need to be explored further for future works in this area. It includes implementation of the framework with proper interfacing between the different modules and implementation of inter-process communication, Expansion of the attacks, in the attack generator, in a general fashion to work with all types of programs , Research into investigating of information needed to determine successful attacks and coverage information in the execution monitor.

7. ACKNOWLEDGMENTS

It gives us a great pleasure to express my deep sense of gratitude and indebtedness to our guide **Mr. Deepak Kumar Gupta**, Associate Professor NIT Jalandhar for their valuable support and encouraging mentality throughout the research work. We also want to express our gratitude to **Mr. Ashwani Malhotra** and **Mr. Amit Kumar,** Assistant Professor NIT Jalandhar for their support.

8. REFERENCES

[1] Alexander Ivanov Sotirov, automatic vulnerability detection using static source code analysis.

[2] Kirill Kononenko, A Unified Approach to Identifying and Healing Vulnerabilities in x86 Machine Code.

[3] David Brumley, Tzi-cker Chiueh, Robert Johnson, RICH: Automatically Protecting Against Integer-Based Vulnerabilities.

[4] Steven Muchnick, Advance compiler design and implementation.

[5] James C Foster, Vitlay Osipov, Nish Bhalla, Niels Heinen, Book on Buffer overflow attack.

[6] R.Bodik, R.Gupta and V.Sarkar. “ABCD: Eliminating array bound checks on demand”. Programming language design and Implementation, 2000.

[7] K. V. N Sunitha, Book on Compiler Construction

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi and D. Jeffrey Ullman, Book on Compilers Principles Techniques And Tools

[9] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer.

[10] M. Gschwind and E. R. Altman. Precise exception semantics in dynamic compilation.