International Journal of Computational Intelligence Systems, Vol. 6, No. 4 (July, 2013), 712-723

A Dynamic Verification Model based on Information Flow Constraint

Dan Wang
College of Computer Science,Beijing University of Technology

Beijing 100124,China

Yan Lu, Lihua Fu, Wenbing Zhao
College of Computer Science,Beijing University of Technology

Beijing 100124,China

E-mail:wangdan@bjut.edu.cn

www.bjut.edu.cn

Received 7 January 2012

Accepted 14 February 2013

Abstract

After analyzing the common attacks for some software systems, a dynamic software behavior verification model related with the unchecked input data based on software analysis and dynamic slicing technology is proposed. Regarding a statement as a basic analysis unit, and the information flow as the main behavior of the software, the direction of the information flow of each statement is defined as its behavior specification, the information flow verification problem is converted into the verification for assigning variable address's validation. During the execution, behavior of the statements that use untrusted variable is monitored to verify whether the address modified by the statements belongs to the specification or not. If it is consistent with the specification, the execution of the statement is permitted. Based on the behavior model proposed, a method of extracting of the behavior specification was researched and a method of dynamic verification was designed. In order to prove for efficiency and performance of the model, the input data related behavior acquiring framework was implemented, and a set of tests were conducted. Preliminary results show the validity of the software's behavior model.

Keywords: program, dynamic, verification, information flow

1. Introduction

Software's behavior may be violated when existence of the vulnerability is triggered by attackers. Much vulnerability in various applications is caused by permitting unchecked input to take control of the application, which an attacker will turn to unexpected purposes. If an attacker tampers with important data of the process using existing vulnerability when the software is running, such as modification of a function's return address, a function pointer, etc., it can interfere with or change the behavior of the software's flow control resulting in damage to the software's normal execution. For example, improper input validation accounts for most security problems in database and web applications.

We first describe three common types of software security attacks aiming to provide a basis for our presented model.

(1)Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability

Copyright: the authors

can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

(2)The format string vulnerability in a program comes from incorrectly using the *printf()* series functions. The attacker can exploit this vulnerability and tamper with any data within its memory address. There are lots of ways to possibly exploit user-controlled format strings in *printf()*. These include buffer overruns by creating a long formatting string (this can result in the attacker having complete control over the program), conversion specifications that use unpassed parameters (causing unexpected data to be inserted), and creating formats which produce totally unanticipated result values. If an attacker controls the formatting information, an attacker can cause all sorts of mischief by carefully selecting the format.

(3)Integer overflow is usually caused by unchecked input data in arithmetic calculation and the results of operations have been used for a number of important operations subsequently, such as memory allocation or cache index, etc., which could be exploited by attackers.

Lack of necessary verification to the input data and protection to information flow in some library functions and pieces of program code is the main cause. When malicious users exploit the software security vulnerabilities to conduct an attack, an abnormal information flow might occur and lead to the variable or address-space information flow inconsistent with their expected use.

The optimal approach to prevent attacks caused by unchecked input data would be to eliminate the vulnerabilities in the affected applications. To this end, an application must properly validate all input data. In order to reduce the code amount that needs to be examined when validating the insecure flow caused by input data, we focus on modifying the outcome of specific conditions by narrowing the search space to include only sensitive input data.

We consider any data that comes from an untrusted source of input as untrusted, such as input from network sockets, since for most programs the network is the most likely vector of attack. We also consider inputs from other sources untrusted, e.g., input data from certain files or *stdin*.

Since input data is usually stored in the form of variables in a program, and its value is gotten by the assignment operation which may cause the information flow. Firstly this paper cites the definition of the information flow in Ref.1, when data value stored in the variable x is spread to the variable y directly or indirectly, e.g. $y = x + 1$, it is considered that there is information flow from variable x to variable y. We regard legal information flow as those assignment operation only to the set of variables and addresses previously-determined by static analysis. Furthermore, we focus on this kind of information flow and regard it as expected behavior in this paper. We first make static analysis of the software source files to obtain expected information flow. Afterwards, during software runs, we track the information flow among statements and verify whether or not the information flow is consistent with expectations. If they are consistent with each other, the current process state can be considered trusted, otherwise, an alarm is triggered.

For ensuring software's behavior to act as its expectation, the goal of this paper is to constructs a software behavior model related with the unchecked input data for identifying and tracking the insecure information flows based on software analysis and dynamic slicing technology. Whenever an attempt to relay on such information is detected, the user is warned and given the possibility to stop the transfer.

Dynamic slicing technology computes a conservative estimate of all statements in a program that are either affected by or affecting the value of a variable at a specific program point and for a given execution, which is the set of statements that propagated information along the illegal flow.

To summarize, the contributions of this paper are as follows:

(1) Constructing an software behavior model related with the unchecked input data by combining software dynamic slicing technology and a complementary static analysis that prevent attacks by monitoring the flow of sensitive information when program executes. Our model need only take into consideration those statements in a program that either directly or indirectly process untrusted input data, so the overhead incurred is reduced.

Co-published by Atlantis Press and Taylor & Francis
Copyright: the authors

(2)The development of input data related behavior acquiring framework capable of performing validation of untrusted data during software executes.

The rest of this paper is structured as follows. Section 2 contains a brief review of some related research. In Section 3, we describe our model and its main components. Section 4 describes the dynamic verification and its implementation issues, respectively. In Section 5 we apply our approach to some executable programs and verify its performance by some common tools. We summarize our plan for future works in Section 6.

2. Related works

There has been some research on software expected-behavior models. In 1996, Forrest et al. proposed a behavior model N-gram[2] based on a short sequence of system calls. Inspired by that, other research based on system calls sprang up. Ref.3 proposed a model combining static analysis and dynamic binding, which had a more powerful capability of detection and lower rates of false alarm. Ref. [4] proposed a model utilizing FSA to construct a calling sequence model, which can describe the structure of loop and branch better. Ref.[5] abstracted system calling sequence and information of context from practicing software over and again, and defined the difference between two different running on information of context as behavior model. It significantly increased accuracy and lowered rates of false alarm compared to N-gram model. Ref.[6] proposed a Control Flow Integrity (CFI) model based on function call relations. CFI constructs function call graph by static analyzing function call relations, and abstracted normal relations as expected behavior model. By rewriting binary execution files, CFI added a piece of codes into function calls and returns jump respectively, then checked the real jump if matched the expected. If not, it recognized the jump as abnormal behavior. Ref.[7] proposed a software information flow expected model. It marks data coming from outside untrusted and traces the transition of that data. According to its defined security strategy, it monitors trust level assigned by the data to protect untrusted data used for address transition, format string, system calling parameters, etc. Ref.[8] also proposed an information flow behavior model which protects software control flow integrity from untrusted data contaminating control data. It ensures software control flow trusted by ensuring the integrity of control data.

This paper considered that making a further analysis on statements or instructions and redefining expected information flow behavior could increase accuracy of description on information flow, and lower miss reports rates during dynamic verification. As our information flow behavior model treated information flow and transition as its behavior, so describing the process by data current state can reflect process's real state accurately and implement dynamic verification, and are able to against a wide range of attacks.

3. Description of expected behavior model

Before further describing our expected behavior model, we give the following suppositions: (1)We suppose that only the verified application program includes an untrusted input and that the input is known beforehand. Other inputs are thought highly trusted. (2) We focus on a program written in *C* language.

3.1. *Some Definitions*

According to the above analysis, the existence of the vulnerability does not affect the normal function of the software unless triggered by the malicious user through the elaborately-designed input data. Therefore, it is not necessary to focus on each of the statements in a program. On the contrary, for reducing overhead we need only take into consideration those statements that either directly or indirectly process untrusted input data. To illustrate this, some definitions are described as follows:

Definition 1 Supposing s_1, s_2 are statements in program *p*. If statement s_1 uses the variable defined in the statement s_2, and the variable in any path from s_2 to s_1 has not been redefined, we define statement s_1 as data dependent on statement s_2, denoted for $s_1 \rightarrow_f s_1$, and $\rightarrow_f$ *will be defined as the transitive and reflexive closure of $\rightarrow_f$. Among them, "defined" refers to the operation of variable assignment. Accordingly, for statement *s* in program *p*, set $FS(s)=\{s_j|s\rightarrow_f{}^*s_j,\forall s_j\in S\}$ denotes the statement set dependent on statement *s* directly or indirectly, where *S* is the set of all statements in program *P*.

Copyright: the authors

Definition 2 Assuming the program *P* has u_{in} which only contain untrusted input data. The statement set that read data from u_{in} in the *P* is denoted as $\{s_1,s_2,..,s_i,..,s_n\}$.According to the definition of *FS(s)*, $FS(u_{in})$ is the statement set in which the statements have referenced untrusted input data in *P* directly or indirectly. According to the related theory of information flow security, for any statements or instructions, if the reference data is untrusted, the defined and generated data by the reference data is also considered untrusted. Because of all the statements in $FS(u_{in})$ have referred untrusted data and they are likely to produce abnormal information flow, the set of $FS(u_{in})$ is the statement set that is urgently in need of verification. For ease of description, the statement in the set of $FS(u_{in})$ is referred to as the concerned statement in this paper.

Definition 3 Assuming the assigned variables in any statement of program *P* is denoted by *def(s)* and program *P* reads data from untrusted input u_{in}, then $FS(u_{in})=\{s_1,s_2,..s_i,..,s_n\}$ is the set of statements that have used untrusted input u_{in} directly or indirectly, the set $\{(s_1,def(s_1)), (s_2,def(s_2)),\ldots,(s_i,def(s_i)),\ldots,(s_n,def(s_n))\}$ is defined as software's expected behavior set.

Definition 4 For each statement s_i of $FS(u_{in})=\{s_1,s_2,...,s_i,...,s_n\}$, if and only if the variable s_i is assigned when the statement belongs to the set of $def(s_i)$, the execution of the s_i in the current context is considered trusted and the current state of the process is considered trusted.

3.2. *Extraction model of the concerned statement*

By applying program static analysis techniques into the source files, we can construct the expected behavior defined above. This process includes two main parts. The first is to extract the concerned statement set, and the second is to analyze the variable set in which each variable may be assigned by one or more statements in the concerned statement set.

As we know, the acquired program slice set by static analysis techniques is the program subset which is composed of the partial statements and the control predicate expressions in the program. Among them, the backward slice's statements and the control predicate expressions affect the definition or the referrence of the value of variable *v* in a certain location *p* of the program directly or indirectly, namely the value of *v* at location *p* depends on the statements of the slice. By contrast, the statements acquired by forward slice techniques and the control predicate expressions rely on the definition the value of variable *v* in a certain location *p* of the program directly or indirectly, tuple <*p,v*> is called the slice criterion, the location *p* is generally expressed by the source filename and the line number. The dependences in the definition of program slicing include control dependence and data dependence.

According to above definition, we use the forward slicing technology to extract our concerned statement set. The example of forward slicing is shown in Fig.1, Fig.1(a) is the source program, Fig.1(b) is its forward slicing set based on the slice criterion <*statement 1, variable n*>.

```
1 scanf("%d",&n);
2 s=0;
3 p=0;
4 while (n>0){
5    s=s+n;
6    p=p*n;
7    n=n−1;
   }
        (a)
```

```
1 scanf("%d",&n);


4 while (n>0){
5    s=s+n;
6    p=p*n;
7    n=n−1;
   }
        (b)
```

Fig.1. An example of forward slicing

Our model needs to remove the untrusted statements and predicate expressions dependent on the untrusted input data in the forward slicing set. We only extract statements possessing data dependence. From the point of view of the algorithm, the program slicing is a graphic accessibility algorithm. Beginning with the slice criterion statements, the algorithm traverses along dependence edges of the data flow between the statements, dependence edges of the control flow and other auxiliary edges of the software system dependence graph (taking statement as node and taking dependence as edge constitute oriented graph) to search statements in the slice. Therefore, by shielding dependence edges of control flow during traverse, we can filter out the control dependence on untrusted input statements and control predicate expressions.

In general, the process is described as follows:

Assume that the program *P* reads data from untrusted input u_{in}, the statement set which reads data from u_{in} in *P* is denoted by $\{s_1,s_2,..s_i..s_n\}$. For any statement s_i

Co-published by Atlantis Press and Taylor & Francis
Copyright: the authors

among them, and variable set which reads data from s_i denotes as $Var_i=\{v_1,v_2,..v_i,...v_m\}$.
The steps of the extraction the concerned statement set are as follows:
(1) Calculating program slicing. For any $v\in Var_i$ taking $<s_i,v>$ as the slice criterion, obtaining the statement set $slicing(s_i,v)$ through forward slice analysis to the source file.
(2) Calculate $FS(s_i)$:
$FS(s_i)=slicing(s_i,v_1)\cup slicing(s_i,v_2)\cup\ldots\cup slicing(s_i,v_m)$.
(3) Calculate $FS(u_{in})$:
$FS(u_{in})=FS(s_1)\cup FS(s_2)\cup\ldots\cup FS(s_n)$.
$FS(u_{in})$ is what we want.

3.3 *Extraction Assigned Variables*

After we get the concerned statements, we need to analyze and obtain the variable set in which each variable may be assigned by one or more statements.
As we know, variables can be divided into global variables and local variables, and variable identifiers cannot be the same within the local scope. Therefore, we describe a variable in the form of <*scope*, *variable identifier*>, and the scope of local variables is expressed by its function name. On the other hand, statements assign the value of variables in two ways: one is the direct assignment, the other is assignment by indirect pointer dereference. For the first way, traversing the source file can determine variable identifiers as well as the scope of a variable. For the second way, variable assignment is in an indirect way through pointer dereference. It is possible that the same pointer in the software code may be used in many places and may point to a number of different variables successively. Therefore, we need analyzing the pointer and calculate the set of variables to which can point and to determine the variables that are assigned through the pointer dereference.
Traversal of the source file can be achieved by traversing its Abstract Syntax Tree(AST). An abstract syntax tree is one kind of middle expression of the program, and can express grammar structure quite intuitively and contain all static information in the source program. It is easy to carry out the traversal and the inquiry to AST.
The analysis process of assigning variables is summarized as follows:
(1) Extract variable identifier of assignment statement from the *AST* including a pointer variable.
(2) Determine the scope of the variable identifier by traversing *AST*.
(3) Use points-to analysis to compute the variable set to which pointer variables may point, and translate the pointer dereference into the corresponding variables set.
Next we mainly discuss the model related with indirect pointer dereference.

3.3.1 *Extract the assigned variable identifier*

Variables can be divided into basic type variables, pointer variables, arrays, structures and other aggregated-type variables. For basic type variables, we extract the variable identifier; for pointer variables we extract the variable identifier and the number of the reference of the pointer's solution; for arrays, structure-type variables, we uniformly extract the variable identifier without distinguishing between the array's elements and structure's member variables.
Program's statements can be divided into three categories. According to the different the types of statement, the correspondent method of the extracting variable identifier is described as follows.
(1)Expression statement. Expression statement is composed of expression and semicolon, the usual form is "expression;". Expression statements are like $a = b +1$; i ++; etc.. After statements are found in *AST*, assignment variable identifier can be extracted by the structure of *AST* directly.
(2)Control statement. Control statements are used to control the execution order of the program and can be divided into following three categories:
① *branch* statement. It includes *if* statement and *switch* statement. Because *if* statements and *switch* statements are used for judging conditional execution, the assigned variable set is empty.
② *loop* statement. It includes *for* statement, *do while* statement and *while* statement. *Do while* statement and *while* statement are conditional judgment statements, so the set of the assigned variable is empty. The form of *for* statement is like (*init,cond,increment*). *init* and *increment* may be any expression statement, but generally only be used to initialize variable of loop control and calculate increment. We suppose *init* and *increment* only contain the related initialization of the

Copyright: the authors

variable of the *loop* control and the increment calculation, therefore variable set assigned by for statement is loop control variable.

③ *jump* statement. It includes *break* statement, *continue* statement, *goto* statement and *return* statement. The assigned variables by *break*, *continue*, *goto* statement is empty set. Accordingly, the assigned variable by *return* statement is an empty set or a temporary variable assigned by the compiler.

(3) Library Function Calling Statement. Calling a library function needs special treatment. Library functions usually transfer pointers as parameters or assign variables by the address of variables, such as output parameters *dest* of the function *strcpy (char* dest,char* src)*. *AST* can only offer variable identifier information which acts as parameters, and it cannot provide information of output parameters and input parameters from parameter list of the library function. Therefore, we introduce the explanatory document for a standard library function, in which the output parameters of the library function are marked out. Combining with the variable identifier that *AST* provides and making use of the statement of the parameter list, we can obtain the assigned variable identifier of the call library information. The parameter list of a library function is described in Table 1. *One* represents output parameters, and *Zero* represents input parameters.

Table 1. express of some output parameters

	Library Function	Statements
1	char* fgets(char* str,int length, FILE* fp)	fgets 100
2	void* memcpy(void* dest,void* source,unsigned count)	memcpy 100
3	int read(int fd,void* buf,int count)	read 010
4	char* strcpy(char* str1,char* str2)	strcpy 10
5	char* strtok(char* str1,char* str2)	strtok 10

3.3.2 *Determine the scope of a variable*

The scope of a variable is divided into global scope and local scope. For the local scope, local variables mask any global variables of the same name. Therefore, from the scope of a function where the statement begins and according to the identifier variable, we will traverse in turn from inside to outside until global scope to search the definition of the variable to determine the scope of the variable. Within this set, the formal parameters of a function and its locally-defined variables are considered to have function scope.

```
int x,y,z;
int* q;
int* m;
int ** p;

1. q = &x;
2. q = malloc(…);
3. p = &m;
4. m = &z;
5. *p = &y;
6. p = &q;
7.  **p = …;
8.  *q = … ;
```

Fig.2. an example of pointer analysis

3.3.3 *Pointer analysis*

Pointer analysis[9,10] is a kind of static analysis technologies. It can calculate the set of storage locations for each pointer variable of source file, including global variables, local variables and the space's dynamic allocation. Andersen algorithm is one of representative pointer analysis algorithms. It is more balanced in terms of accuracy and efficiency and suitable for analyzing large-scale software. Therefore, we chose it to analyze pointer variables. For the fragment of program in Fig.2, in statement 8, the Andersen algorithm calculates pointer p, q. For pointer variable *p*, its pointed variable set is {*x*,*y*,*malloc_6*}; For pointer variable *q*, its pointed variable set is {*y*};

We assume that the value returned by the function *malloc()* is a dynamically allocated heap address. Because some applications use pointers that can have multiple targets, we need to analyze these multiple-target pointers. Our analysis method is described as follows: assuming that program has a pointer variable *p*, *p* points to the set of variable $pts(p) = \{p_1,p_2 \ldots p_i \ldots p_n\}$, among them, $p_1,p_2 \ldots p_i \ldots p_n$ are all pointer variables. When the statement *s* assigns to $**p$, then the variable set that *s* can assign is $pts(p_1) \cup pts(p_2) \ldots \cup pts(p_n)$.The assignment of double pointer variable is shown in the

Copyright: the authors

```
int x,y,z;
int* q;
int* m;
int ** p;

1. q = &x;
2. q = malloc(⋯);
3. p = &m;
4. m = &z;
5. *p = &y;
6. p = &q;
7.  **p = ⋯;
8.  *q = ⋯ ;
```

Fig.3. Multi-pointer analysis

fragment of program in Fig.3. After the analysis of Andersen algorithm, pointers *m*, *p*, *q* points to a set of variables. For variable *m*, its pointed variable set is {*z,y*}; For variable *q*, its pointed variable set is {*x,malloc_2,y*}; For variable *p*, its pointed variable set is {*m,q*}. In statement 6, pointer *p* points to a set {*m,q*}, therefore, when assigned to ***p*,we can use the set $pts(m) \cup pts(q)$,namely,{*x,malloc_2,y,z*}.

3.3.4 *Model optimization*

We should try to minimize the time and space overhead in the phase of dynamic verification. The results of existing static analysis show that we can acquire the safety statements (statements not vulnerable to malicious intrusion) in the program through static analysis. If a statement assigning a variable meets (1) and (2) of the following requirements simultaneously, or meet only (3) we believe that the statement is safe:

(1) It only writes into a fixed address. Address written by statement is fixed rather than calculated dynamically. There are two types of fixed addresses, one is fixed logical address, such as global variables of basic types; another is fixed offset address, such as local variables of basic types.

(2) It only writes a fixed number of bytes. The statement may write only a finite number of bytes of data. For example, the assignment of variable type belongs to *integer*, *long integer* and so on.

(3)Temporary variable. The assigned variable is a temporary variable that the compiler allocates.

Based on the above three conditions, our optimized strategy is to remove the following two types of statements from the concerned set of statements:

(1) Control statements. According to the above analysis, the set of variable which a control statement can assign is an empty set or only contains the loop control variable or temporary variable, and the loop control variable is assumed to be of a basic type, either global or local variable, so we can think of these as safe statements.

(2) Expression statements which assign variables of basic types directly, such as *a=b+1*,*i++*. They accord with the above safety conditions and don't produce abnormal information flow, so we can consider they are safe statements.

3.3.5 *Variable address convention*

The variable extracted in the previous section is expressed as the form of *<scope,variable identifier>*.However, it is based on source-level representation, and cannot be directly used for dynamic verification. In order to achieve dynamic verification easily, source-level representation needs to be converted into actual memory addresses. Global and local variables have been allocated addresses at compile time, so the address can be obtained from the debugging information directly. Heap space isn't allocated address at compile time, but is assigned addresses dynamically by address of functions call such as *malloc()*. The detailed information is described as follows.

(1)Global variable. Expressed as *<address*,*count>*, *address* is the logical address of a variable and *count* is the length in bytes.

(2)Local variable. Expressed as *<fun*,*offset*,*count>*. *fun* is the scope of a variable expressed as the function's name;*offset* is the offset of variable related to the stack pointer (ebp) of function *fun ()*,*count* is the length in bytes.

(3)Heap space. Expressed as *<fun*,*call_offset>*. *fun* is a function which includes the call statement of *malloc()*; *call_offset* is the offset of *malloc()*'s call instruction relative to the first address of function *fun ()*.

3.3.6 *Writing statement and its convention*

As described above, the statement is on source-level representation and cannot be directly used for dynamic

verification. Source-level representation needs to be converted into corresponding statements and expressed by its address. What we are concerned with in this paper is the variable assignment behavior, so we are concerned only with those statements which write data into memory. These statements are called writing instructions here. Therefore, we need to analyze writing statements and their corresponding addresses taken from debugging information. First, we locate a group of instructions corresponding to concerned statements after compilation, then filter out other types of instructions and retain the writing instructions and call to the standard library. Finally, we extract their address.

Writing instructions and call instructions to library functions are expressed as *<fun,offset>*, *fun* is a function which contains the instruction, *offset* is the offset address of the instruction relative to the first address of function *fun()*'s code.

4. The dynamic verification

After obtaining the model of the software's expected behavior, it is necessary to conduct verification dynamically during runtime. Monitoring the actual behavior of software through obtaining the assignment behavior of concerned statements and extracting its assignment memory address, then comparing them with the corresponding expected statements' behavior are main tasks. If observed behavior is consistent with expected one, we believe that the current state of the process is trusted, otherwise it's untrusted. However, there are two difficulties in realizing measurement.

(1) Determining verification point. Verification point is the location or opportunity to verify the behavior of software during runtime. We focus on writing instructions and call instructions of library functions in this paper.

(2) Determining maintenance points. The address of local variables changes constantly with the execution of *call* and *exit* function. The address may be dynamically allocated and released continually. Therefore, along with the running of software, we must update the expected set of address in a timely manner and ensure the correctness of expected data. The location or time of updating and maintaining a variable's address is referred to as expected data maintenance point in this paper.

4.1 *Maintenance for expected data*

Initially, we set required expected data maintenance points according to the information of variables which can be assigned in the expected behavior. If the software encounters an expected data maintenance point when running, it will update the address information accordingly.

Maintenance points for expected data include the following categories.

(1) *call* and *exit* of function. During software runtime, the expected address of a local variable needs to be updated when function is invoked or exits.

As mentioned above, expected assigned local variables can be expressed as the form of *<fun,offset,count>*, *fun* is a function that defines variables. For any local variable *v<fun,offset,count>* in the expectation set, the address of variable *v* in the expected set needs to be reset when function *fun()* is called. When *fun()* exits, the address range of variable *v* needs to be removed from the expected set.

(2) Allocation and release of heap space. During software runtime, heap space is allocated and released through calling *malloc()* and *free()*. Similarly, *malloc()* is expressed in the form of address*<fun,call_offset>*, so the expected address set needs to be updated after *malloc()* located in address *<fun,call_offset>* executes. The release of address space needs to be removed from expected address set on return from function *free ()*.

(3) The loading of dynamic link library. The software cannot do without the support of libraries. In addition to visiting its own local variables, it also visits the dynamic distribution of heap space; part of the link library also defines global variables. They are stored in the *.bss* section or *.data* section of link library. As we cannot analyze the source code of library functions in the static analysis phase, we do not know whether global variables are assigned by library functions. According to the read-write right of the *.bss* segment and *.data* segment, our model considers the updated address range belonging to segment as the address that can be assigned of library functions. When the libraries are loaded and linked into the process space, their address range of sections is added to address set that can be assigned by library functions. This paper assumes that the program only links the runtime library *glibc* of C language.

Copyright: the authors

4.2 *Address Verification*

When the monitored software is executed to a verification point, it will extract the writing memory address of program and verify whether or not the address belongs to an expected address set. As mentioned above, verification points of information flow's behavior include two types of instructions.

(1) For writing instructions. When a writing instruction is executed, it dynamically obtains the actual memory address and byte count written by the instruction, and finds the entry in the memory address set that the instruction can be assigned. If the actual writing address does not belong to any address range of set, we believe that it is inconsistent with expectations.

(2)For the function call instructions. Because we are unable to analyze internal assignment behavior of library functions, we consider library functions as a whole. We do not check their assignment behavior for internal variables and only ensure that other variables aren't written into the process illegally. During the running process of library functions, all written addresses during recording are verified after returning from library functions. The addresses written by the library functions can be divided into the following categories:

(1)The local variable address of library functions. If writing address is the local variable address of library functions or belongs to the local variable address of function call in its internal implementation, we will believe that the assignment is legal and doesn't verify any more.

(2) The address of the heap area. If the writing address belongs to an allocated address in the heap, we will first check whether the address belongs to the allocated heap space of application-defined functions. If not, it indicates that it is the internal heap space of library function, and we believe that the assignment is legal. Otherwise, we will find in the address set that can be assigned in the measurement points and verify whether the writing to heap space belongs to the expected address set.

(3) Other Address. If the writing address doesn't belong to the address of above (1)(2), for example, the global variables of process, the local variables of application-defined functions and so on, we will scan the address set that can be assigned in the verification points and verify whether the address belongs to the expected address set.

5. Analysis and Tests

This paper performed the test and evaluated on several software implemented in the C programming language under the Linux operating system, kernel version 2.6.18. For the sake of simplicity, accuracy as well as effectiveness, this paper utilized the tool *ROSE*[10] to construct the expected behavior model. Meanwhile, we used dynamic instrumentation tool *Dyninst*[11] to achieve dynamic verification of software behavior during runtime. Fig.4 shows the main modules of our test framework. There are two main parts including static analysis and dynamic measurement. The related modules of static analysis analyze source files and construct expected behavior based on feedback from *ROSE* which outputs software's expected behavior and stores it into files. In the meantime, we developed dynamic instrumentation tools by combining the *Dyninst* which injects codes into target codes to monitor

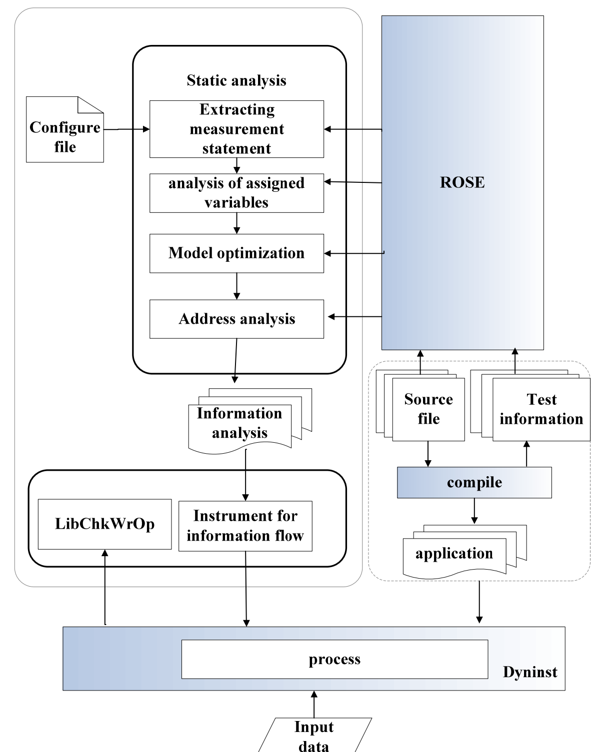

Fig.4. Verification system components

Copyright: the authors

the execution of process. The function of each module is described as follows.

(1) Extract measurement statements module. It reads untrusted input statements from configuration files and utilizes the program slicing techniques to extract statements that directly or indirectly rely on the untrusted statement set.

(2) Analyze assignment operation module. It reads the variable identifier of assignment operation from the abstract syntax tree, and then analyzes the scope of the identifier to identify the variable. By pointer analysis, it calculates each possible variable set that can be referenced by the pointer, and then converts pointer dereference to actual variables.

(3) Optimization module. It traverses statements associated with abstract syntax trees and determines the secure statement, and its assignable variables set. Only control statements and expressions that make assignment to basic types are removed by this process.

(4) Analyze address module. It converts statements and variables of source codes into statements and address respectively through reading *DWARF*[12](Debugging With Attributed Record Formats) by use of *ROSE*.

(5) Information flow behavior instrumentation module. It loads process by means of tool *Dyninst* and analyzes expected behavior of software information flow, then locates verification points and data maintenance points of executable files by calling *Dyninst.* Then, it injects address authentication functions into verification points and maintains expected data functions on maintenance points, respectively. After configuration, it calls the *Dyninst* interface to start the process and monitors software runtime.

(6) *LibChkWrOp*: *LibChkWrOp* is a dynamic link library of *C* programming language. A series of functions in the library implement expected data management, address authentication, initialization, etc.

We tested the expected behavior model by using some test programs with artificial security vulnerabilities including stack buffer overflow, heap buffer overflow and format string attack. Fig.5 shows our test program. It treated inputs from *stdin* as untrusted data.

(1) Stack buffer overflow. In the code fragment shown in Fig.5 (a), statement 3 had a vulnerability of stack buffer overflow. If it inputs 18 bytes into *buf* and covers the return address of *foo()* with the entrance address of *fun()*, the function *foo()* would be unable to return to its correct address after execution.

(2) Heap buffer overflow. In the code fragment shown in Fig.5 (b), statement 7 had a vulnerability of heapbuffer overflow. If it inputs 64 bytes into statement 7, and writes address of *fun()* into the four bytes between (*p*+60) and (*p*+63), writing the value of ((*&f*)-12) into the four bytes between (*p*+56) and (*p*+59), respectively when program executes to statement 8, the buffer "*p*" would be released. As addresses "*p*+56" and "*p*+60" happened to be the pointer address of heap free fields doubly linked list, so when free field merged after "*p*" released, "*(((*&f*)-12)+12)=(*fun*)" equals "*f*=*fun*", pointer "*f*" has been tampered. When execution reaches statement 9, the program's executing flow of process has been violated.

(3) Format string attacks. In the code segment shown in Fig.5 (c), statement 6 had a vulnerability of format strings. The address of global variable *i* is "0x8049604". If we input string "\x04\x96\x04\08%d%d%d%d%n"

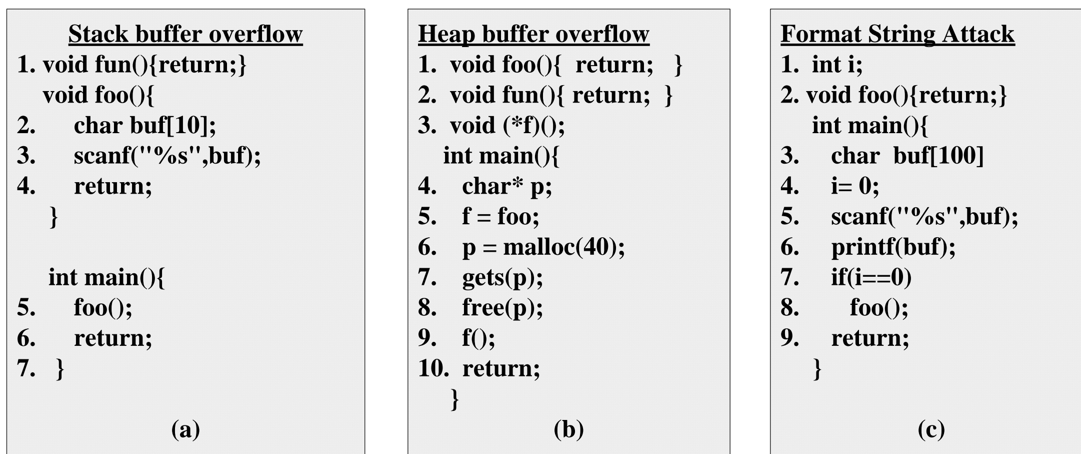

Fig.5. Some Test

Copyright: the authors

into *buf* via function *scanf()* at statement 5, then while performing function *printf()* at instruction 6, the value of *i* would be tampered as call statement 8, and branch statement 7 would be tampered subsequently.

Test results showed that our model could detect abnormal information flow in functions *scanf()*, *gets()* and *printf()*. Furthermore, we tested model effectiveness with two practical examples, which treated remote input as untrusted:

(1) *wu-ftpd* format string vulnerability. *wu-ftpd* is a commonly used server on Linux, and offers basic and simple ftp service. In version 2.6.0 or earlier of *wu-ftpd* server, it had the format vulnerabilities on calling function "*vsnprintf()*". Attackers could use the vulnerability to get super-user permissions by rewriting user login ID, or to tamper with function return addresses to change program control flow. Our test tampered function return address.

(2) *Openssh* integer overflow vulnerability. *Openssh* is a set of connection tools for safe access to remote computers. Before version of 2.9, *Openssh* has an integer vulnerability in the process of authenticating remote accesses. We used a 32-bit integer to assign a 16-bit integer variable which is the parameter of function *malloc()*.After the variable overflowing by malicious input, attackers can tamper data in any address. Our test used this vulnerability to tamper the value of decision variables of branch statements in authentication function, and that would allow users attempting a connection to avoid the security authentication mechanism.

Test results showed that our expected behavior model did detect the two security attacks above.

6. Conclusion and future works

We propose a model to verify whether the program follows its expected behavior during execution. Aiming at its limitation, we plan to investigate the following aspects in the future:

(1)This paper treats aggregate-type variables as a single unit during assignable-variables collection and statement analysis and does not distinguish members of aggregated types. This strategy is far easier to implement, but leads to inaccuracy, which could cause miss-reports. How to increase accuracy and lower miss-reports is a topic for future study.

(2)During the analysis period, we fail to analyze assignment behavior of library functions effectively for lack of source files for library functions. Therefore, the default assumptions of assignment to internal variables of library functions are correct, which could also cause miss-reports. At the same time, this paper relies on pointer analysis techniques to construct information flow expected behavior. For the sake of execution effectiveness, the pointer analysis techniques might cause some loss of accuracy, the reported results are generally larger than actual one because it enlarges the range of assignable addresses. That could also cause miss-reports. We will give further research on these issues a high priority.

(3)Follow-up research will conduct validation experiments on larger data sets.

Acknowledgment

This work is partially supported by Beijing Municipal Natural Science Foundation of China under Grant No.4122007.

References

1. L.O. Andersen. Program Analysis and Specialization for the C Programming Language, *Copenhagen:University of Copenhagen. Ph.D.* 1994,pp.112-152.
2. S. Forrest, S.A.Hofmeyr, A.Somayaji, T.A.Longstaff. A Sense of Self for UNIX Processes. *In Proc. of the 1996 IEEE Symposium on Security and Privacy*. Los Alamitos, CA(IEEE Computer Society Press. 1996) pp.120-128.
3. W.Li,Y.X.Dai,Y.F.Lian. Context sensitive Host-based IDS using Hybrid Automaton. *Journal of Software*. 20(1) (2009),pp.138-151.
4. C.Michael, A.Ghosh. Using Finite Automate to Mine Execution Data for Intrusion Detection: A Preliminary Report. *Lecture Notes in Computer Science (1907)*, (RAID 2000),pp.66-79.
5. H.Feng, O.Kolesnikov, P.Fogla, W.Lee, W.Gong. Anomaly Detection Using Call Stack Information. *In IEEE Symposium on Security and Privacy*, Oakland, California(2003),pp.62-76.
6. M.Abadi, M.Budiu, Ú.Erlingsson. Control-flow integrity. *In Proc. of the 12th ACM conference on*

Copyright: the authors

Computer and communications security, New York, USA(2005),pp.340-353.

7. J.Newsome. D.Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *In Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)* (2005),pp.10-17.
8. R.Jedidiah. Crandall, T.C.Frederic. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *In Proc. of the 37th annual IEEE/ACM International Symposium on Microarchitecture*(2004),pp.221-232.
9. S.Horwitz, T.Reps, D.Binkley. Interprocedural slicing using dependence graphs. *ACM SIGPLAN Notices*. 39(4) (2004),pp.229-243.
10. N.Wang,J.Liu. Proficiency and effectiveness comparision of five types pointer analysis algorithm. *Computer Engineer and design*. 24(12)(2003),pp.38-42.
11. ROSE. https:// www.rosecompiler.org/.
12. Dyninst. http:// www.dyninst.org.
13. DWARF. http://www.dwarfstd.org/

Co-published by Atlantis Press and Taylor & Francis
Copyright: the authors