Behavior Analysis-Based Dynamic Trust Measurement Model

Dan Wang, Xiaodong Zhou, and Wenbing Zhao

College of Computer, Beijing University of Technology, Beijing 100124, China
wangdan@bjut.edu.cn

Abstract. The trust of an entity is based on its behavior's trust in trusted computing technology, and software's trust can be measured dynamically by its behavior when it is executing. However, conducting dynamic measurement is a big challenge. Defining and building software's behavior is the basic work of measuring software trust. A behavior-based dynamic measurement model for an execution program is provided, which applies the method of describing program behavior by control flow graph to dynamic trust measurement. The model first measures the program before it is loaded, then generates the expected behavior model of the program according to static analysis. Then, the model monitors the program's execution in real time by verifying the flow branches of the program with the expected behavior model. Finally, the paper analyzes the security of this model and indicates that this model is able to protect against some code-injection attacks which can't be handled by the traditional static measurement method.

Keywords: Program, dynamic, trust measurement, behavior analysis.

1 Introduction

Trusted computing technology has become a research hotspot in the computer security area. If a software's behavior and running result can meet its user's expectation and can provide continuous service when interfered, it can be regarded as a kind of trusted software[1]. Defining and building software's behavior is the basic work of measuring software trustworthiness

At present, the trust measurement technology is based on the integrity measurement. There are two mechanisms that use integrity measurement technology in the system boot process. One is the secure boot[2][3] mechanism which is designed to verify the integrity of the booting process and stop the boot process when the integrity measurement fails. The other mechanism is called trusted boot[4]. Unlike secure boot, this mechanism only takes measurements and leaves it up to the remote party to determine the system's trust.

However, during the period of the operating system kernel loading and the users' applications executing, there are many problems such as the users' uncertain operation to the system, and the huge variety of user applications. There are also some security problems during the program's execution period such as follows:

S. Qing et al. (Eds.): ICICS 2011, LNCS 7043, pp. 267–281, 2011.

- The intruders may get the root privilege of the system illegally and then modify the program without the user's notification, and then the modified programs are executed by the user.
- System users download and install untrusted software.
- Many applications have code injection (such as buffer overflow) vulnerabilities. In these cases, intruders can alter the behavior and the execution flow of the program when they find that such program has this kind of vulnerability.
- The intruders get the root privilege of the system illegally and then modify the kernel of the system. In Linux, the root user can modify the kernel data by loading and unloading kernel modules.

1.1 Integrity Measurement

The integrity measurement technology before the program loading is able to resolve the first problem well. Before the program to execute, the integrity measurement component generates the hash digest with certain algorithm, and compares it with the original digest. Then the integrity measurement component can make sure the information of the software is not tampered just before it is loaded. The software must acquire certain authorization and execute under these authorization only it is measured to be trust. Therefore, integrity measurement can find whether the software is tampered or not (but still can't prevent the intruder to get the root privilege), and the tampered program can't execute. However, this kind of integrity measurement process is not a dynamic one which can take trust measurement during the program's execution period, such as the code injection attack as problem 3 mentioned above.

1.2 Related Works

The research literature indicates the presence of many efforts in the area of software integrity measurement. The Linux Integrity Measurement Architecture (IMA)[5] can measure code loaded and static date files such as configurations used, such that a remote party can verify that a Linux system contains no low integrity components. PRIMA[6] is based on IMA and can minimize the performance impact on the system. With PRIMA the number of measurement targets can be reduced to those that have information flows to trusted objects. The BIND[7] system measure discrete computation steps by their inputs and code. But the integrity measurement technology still can't well resolve the time-of-measurement and time-of-used problem and their measurements granularity are in file level. The integrity measurement only reflects the memory state right after the program is loaded but the software may be compromised at run time. Reference [8-10] proposed a model combined static analysis and dynamic binding, which had a more powerful capability of detection and lower rates of false alarm. Some research used system calls came out from software runtime to construct behavior models.Systrace[11] is a computer security utility which limits an application's access to the system by enforcing access policies for system calls. It was developed by Niels Provos and runs on various Unix-like operating systems.Systrace is particularly useful when running untrusted or binary-only applications and provides facilities for privilege elevation on a system call basis, helping to eliminate the need for potentially dangerous setuid programs. Promon's

Integrated Application Protection (IAP)[12] makes applications tamper proof and ensures the privacy of all associated application data. It is offered to all software application providers that have a need to protect sensitive information and ensure application integrity. Promon IAP is integrated into the applications as a separate compiled module which interacts with the core application through an API. This raises the security level of a single application without influencing the main system.

1.3 Contributions

In this paper, we focus on the behavior of the software when it is executing and we propose a model that can measure the program's trust by determining if the program follows its expect behavior during their execution. We treat the control flow graph of the executable as its expected behavior. We first generate the program's expected behavior by static analyze the binary code of the executable program, and then we monitor the program's execution within the control flow graph.

Our behavior-based dynamic trust measurement model for an executable program focuses on the problem 3 mentioned above. Our model is a dynamic measurement model based on the integrity measurement. But our architecture does not guarantee that an attacker cannot obtain root privileges, which is beyond the scope of this paper.

The rest of this paper is organized as follows. Section 2 contains a description of the expected behavior. In Section 3, we describe our model architecture and its main parts. Section 4 describes our system's implementation by using the NFA model. In Section 5 we apply our approach to some executable program and verify its performance by Pin tool. We summarize our plan for future works in Section 6.

2 Expected Behavior Description

To date, there have been many related researches focusing on how to describe the software's behavior. And the most representative approach to describe the software's behavior is the system-call sequences generated during the program's execution. In the Linux operating system, system call is the only method for the kernel to provide services to the user applications. The user applications use all kinds of system resources by trapping into kernel mode from user mode, and system calls are the only mechanism for this interface. Many important operations such as read or write of files, creating processes and so on are all implemented by system calls. So the system call sequences generated during the program's execution can be used to describe the behavior of the program to a certain extent.

Currently there are two major methods for modeling the behavior of a program through the tracking of system call sequences. One is the dynamic learning method[13], and the other is the static analysis method[14][15][16]. The dynamic learning method can capture the run-time data on system call sequences and timing over many program executions within a safe and controlled system environment. The behavioral model for the program is based on the statistical data for the captured sequences. However, regardless of the number of times the program runs under these conditions, the model cannot reflect all possible paths the program traces during

execution. Consequently, we believe that the method of dynamic learning to capture the system call sequences produces an insufficiently complete model of the software, and is therefore unsuitable for dynamic trust measurement.

In contrast, the static analysis modeling method which statically analyzes the source or the binary code of the program is capable of capturing the whole flow graph of the execution. Thus, it can well reflect the expected behavior of the software. More specifically, an analysis of the binary program code is more efficient than an analysis of the source code for two reasons. First, the source code for most software is difficult to obtain, and second the binary provides a better representation of the program's behavior on the specific hardware platform. So the static analysis of the binary software code is more general.

We statically analyze the binary code of a program to generate the flow graph. Then we use this graph-based model to describe the dynamic behavior of the program.

3 Our Model

Before we describe our model, we first make some assumptions about the system: (1) all programs are obtained from trust sources. The digital signature of the software is also obtained, accompanied by the software itself. (2) The system is implemented based on secure boot, so the boot processes are not tampered with. The first assumption means the content(code files) of the program must not be tampered with before the issue of dynamically trust of a running program to be discussed. The second assumption means that the trust of the program on application level is based on the trust of the lower level program. These two assumptions are the basis of the application's trust.

At first, we present a simple stack overflow attack example and analyze the software is how to change its behavior from the control flow perspective.The example is shown in Fig.1.

```
Int single_source(char *frame)
{
Char buf[256];
FILE *src;
Src=fopen(fname,"rt");
While(fgets(bug,1044,src)){
……
}
Return 0;
```

Fig. 1. An example of a Stack overflow attack

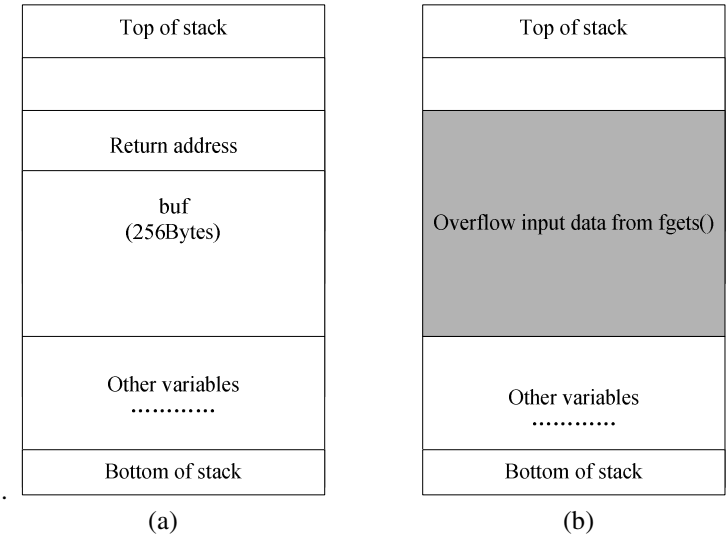

Fig. 2. Stack states comparison

The program above reads data from files line by line and stores them into buffer buf. The initial stack state of the program is shown in Fig.2(a). The stack state is shown in Fig.2(b) after execution. So attackers can overlay the return address of buffer with any data in order to change the control flow of the program.

When a program is running, the instructions which can change its control flow include CALL, JMP, JXX (JNZ, JE, etc.), RET. The changes of the control flow during program running can be seen as the behavior of the running program. A program always runs from a fixed code segment. Our model sets a corresponding initial state. When a program is running, it is considered that a new state is transferred into every time the program's control flow changes. Therefore, it is feasible to acquire a program's expected behavior before starting it by analyzing its control flow from the instruction code. Then, when the program is running, whether its behavior corresponds with the expected one can be judged.

3.1 Model structure

The structure of our model is shown in Fig.3. The model is based on the integrity measurement. First the software would be verified its integrity according to the integrity check process, then the binary code of the software is verified to make the expected behavior model. At last, when the software is being executed, the execution trace of the software is verified whether it followed the model or not. It is a dynamic process.

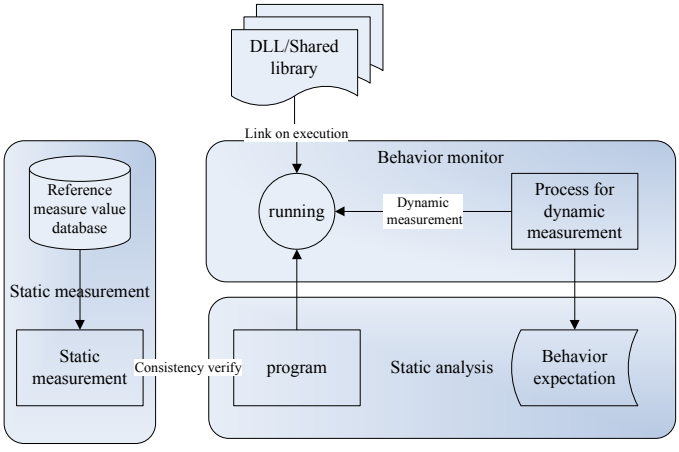

Fig. 3. Behavior-based dynamic measurement model

3.2 Main Components

Our model is constituted with a static measurement component, a static analysis component, and an expected behavior monitor.

(1) Static measurement component: Before the program's execution, this component applies integrity measurement to the executable and other relevant files. A program which passes the integrity measurement can be loaded to execute. The static measurement component guarantees the integrity of the program. A program not only executes the code itself, in most cases, the program needs the support of many other files (such as the library, configuration files and so on). So the static measurement component can't measure the integrity of the executable only, but also needs to measure the integrity of these support files. For the lib files needed by the program, under the static link situation, all the lib code is linked into the executable, so the integrity of the executable itself can be directly measured. But under the dynamic link situation, all the dynamic link libraries used by the program should also be measured for their integrity. For example, in the Linux system, the identification information for all the shared objects accessed by the program can obtained from the head section of the ELF format executable. So it is feasible to measure the integrity of the shared objects.

(2) Static analysis component: The static analysis component statically analyzes the executable library code in the system to form the expected behavior model. The dynamic library code can be analyzed early and the result saved to disk. When the program accesses one of these shared objects, the object's model can be merged with the model of the executable program to form an integrated model. Because the expected behavior model is saved to files, the integrity of these data files should also be guaranteed.

(3) Expected behavior monitor. The program which passes the integrity measurement will execute under the expected behavior monitor. The monitor will get the current state of the execution, and determine if the state is in the expected

behavior model generated by the static analysis component. If the program enters an unexpected state according to the behavior model, the monitor will decide that the program does not follow the expected behavior and terminate the execution.

3.3 Data Structure Design

As described above, the dynamic measurement is executed in the process of program running. Therefore, our experiment is correspondingly divided into two steps: static analysis and dynamic measurement.

The analyzed object is the executable instructions during the static analysis. Due to a number of potential different execution paths of a binary, we use a static link to refer to the actual executable file of our experiments.

Since our model regards the jump among the control flow during the program execution as the state transfer, the corresponding data structure to represent the program's state is designed and shown in Fig.4. The whole state graph is a graph structure, each node represents a state, which contains start address, next state pointer while existing sate transfer and next state pointer without sate transfer.

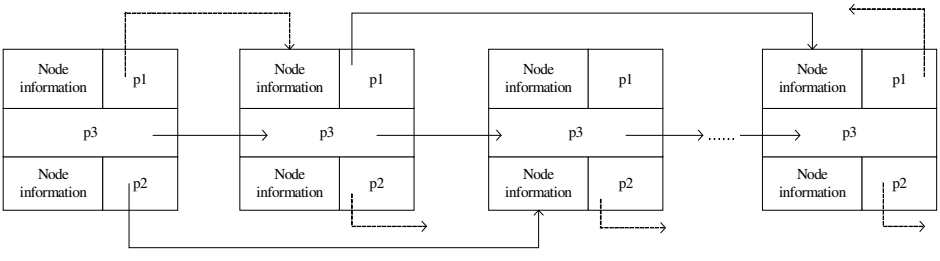

Fig. 4. State graph data structure

In Fig.4, pointer p1 will point to the next right state when sate transfer happens. When Call or Jump instruction executes, pointer p1 will point to the state which represents the target address. Pointer p2 will point to the next state when there isn't sate transfer. For example, when the condition jump instruction executes, the next sate should be the state which p1 points to if the condition is true. On the other hand, if the condition is false, the next sate should be the state which p2 points to. All states will be linked together by pointer p3. The states among the linked stated by p3 don't have jump relation. Pointer p3 is assigned a value after the first scan. The node information field of the state node saves the start address of the instruction segment and other information.

After making static analysis to the executable file, each point field of the node in sate graph is assigned a value. The first node in state graph is the program's first sate. Program executes from the first state. As we know, program always executes from a fixed starting address which is called the entrance function, but it isn't the main() function. This entrance function has different name and it depends on the platform. After the OS creates a process, the control is passed on to the entrance function, which is always to run certain function in its lib. This entrance function initializes the runtime lib and runtime environment including heap, I/O, thread, global variant, etc.

As soon as this initialization process is completed by the entrance function, the main() function can be invoked. Since the entrance function of the lib glibc is _start on Linux platform, this _start function later invokes __lib_start_main, therefore in state graph, the first sate node represent _start function. After main() function executes, it will return to the entrance function and do some clear work such as to destruct the global variants, to destroy heap, close I/O, etc. Later, it terminates the process by executing certain system call. Consequently, the last node in state graph represents the last function of entrance function to do clear work.

4 Implementations

The technology of statically analyzing the binary code of the program has been used in some intrusion detection researches. For modeling the executable and library code, there are two main methods. One is the context-insensitive non-Deterministic Finite State Automaton (NFA) which neglects the return information of functions. The other is the context-sensitive pushdown automaton (PDA) which records all the return addresses of functions with a stack. When a function returns, the return state is determinate according to the top of the stack. Both of these two models are merged by local function automaton. The NFA model will add a transition directly from the call state to the target state. When the function returns, another transition will be added from the return state to the call state's next state. Because a function may be called more than one time from different addresses, there will be several transitions from the return state. The PDA model uses a stack structure to remember the exactly return address of one function call. When the function call returns, the top of the stack is the return state for that call and the transition is definite.

Because the NFA model is much more efficient, our implementation is based on the NFA model. First we statically analyze every function's binary code and generate the local function control flow graph (CFG). The CFG of the program consists of many basic blocks and the transition among these basic blocks. The branch is based on the JMP instruction and the CALL instruction. The state of the program in our model is based on these basic blocks. For the CALL or the conditional JMP instruction (such as JNE, JE and so on), two new states will be created if they aren't in the states set. One is the state reflecting the target address, the other is the state reflecting the next instruction just below the current instruction. For the JMP instruction, only the target state will be created. Fig.5 shows the states in the control flow graph according to the branch or call instructions. These branches or call instructions make the control flow of the execution change. We treat a set of instructions of the executable which has no control flow transfer in it as a basic block. And we also treat the one basic block as one state of the execution.

One of the most important attributes of a state is the address of the first instruction of the state's related basic blocks. When the monitor component observes a transfer, the state of the execution will be changed. The static analysis is taken by two steps. In the first step, we statically analyze the whole binary code and find out all possible states into which an execution may transfer. In the second step, we analyze the executable again and for every possible state, find out all its possible next state. As a result, a control flow graph with numbers of states (basic blocks) and transitions is created.

When we monitor the program's execution, we dynamically capture every instruction in the trace of that execution. At the beginning, the state of the program is in an init state of the execution. Every executable begins its execution from an init routine instead of main function. When an instruction is executed, the monitor makes a decision based on the state of that instruction. If the address of that instruction is a target address of a call or jump instruction, the instruction must be in a new state. If this situation happens, the current state of this execution will transfer to the new state.

Because all the possible states and all the possible transitions have been obtained by the two-step static analysis, if the current state of one time execution doesn't transfer into the approved state, the program's behavior is not following the expected one.

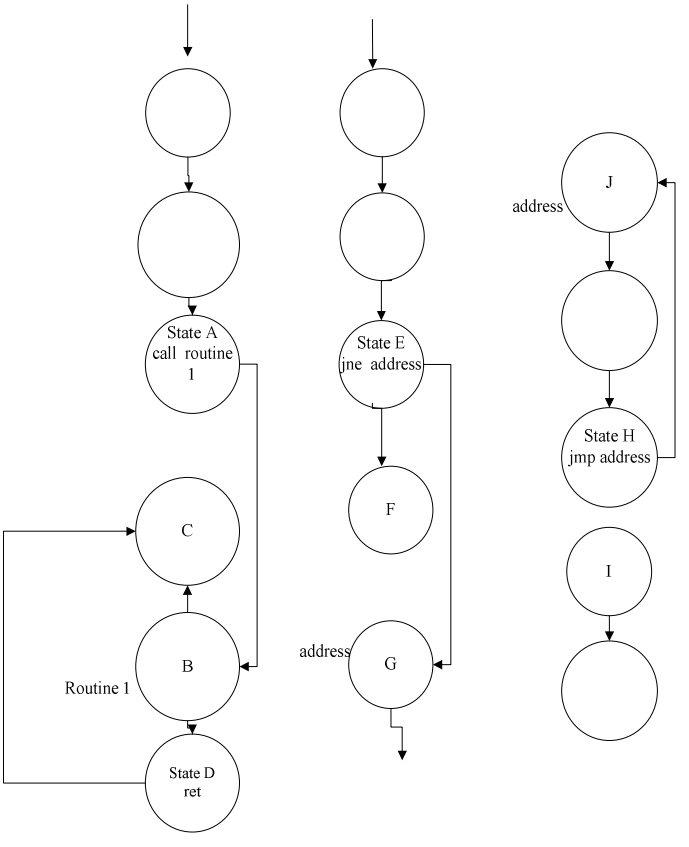

Fig. 5. States according to branch or call instructions

Then we use a kind of common technique used to construct a system control flow graph, the call site replacement, as shown in Fig.6. We replace every edge representing a function call with the related local call graph. At this point, the whole

control flow graph of the program based on the NFA model is complete. If the $G =< V, E >$ is the control flow graph of the program, V is the set of basic block, E is the set of the edges among V ,according to the control flow graph, every basic block is a state, and every edge can be treated as an input symbol table. Then the NFA model can be described as follows:

$P = (Q, \Sigma, \delta, q_0, F)$, Q is the states set, Σ is the input alphabet, $\Sigma = (call + address \cup jump + address)$, $call$ is the function call instruction and $jump$ is the branch instruction including JMP or JE and so on. $address$ is the related instruction address ; q_0 is first state of the automaton, and is the only entry of the CFG ; F is the set of accepting states, that is, the exit basic blocks of the program. δ is the transition relation and $\delta(q_1, a) = q_2$, $a \in \Sigma$ means state q_1 transport to state q_2 according a .

Clearly, the JMP and the CALL instructions will cause a state transition of the program. But there is another situation where one state can transfer control to another without a CALL or JUMP instruction. When the instructions are executed fall through, the next instruction that will be executed may be a target of some CALL or JUMP instruction. This instruction is a beginning of a new state. In this situation, states transfer also happens.

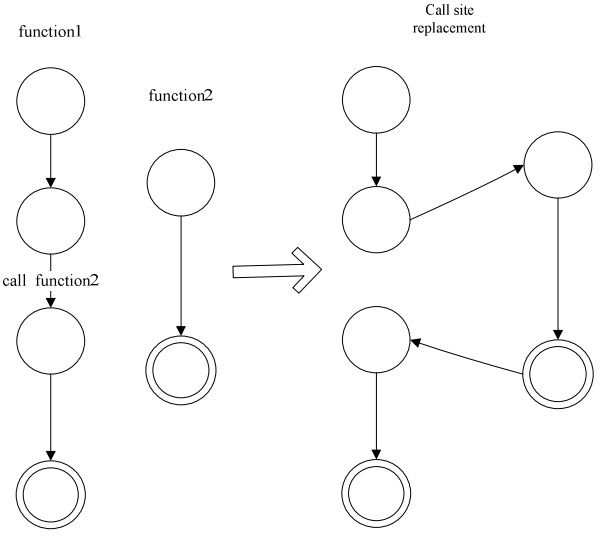

Fig. 6. The merger of the function automaton

After getting the expect behavior model of the program, the expected behavior monitor will do the dynamic measurement to the program when it is executing. The measurement arithmetic is shown in Fig.7.

```
InputStateGraph(stateGraph);     //Input the state graph
State currentState = stateGraph.firstState();
 //set the current state as the first state
for every instruction ins that is executed
 do
    State state = stateGraph.getState(ins);
    if(state exists)    // a new state is trasferred to
        if(currentState.nextState()==state)
 // transtition exists between the two states
             currentState=state;
             continue;
        else              //or the program is untrusted
            untrusted;
        end if
    else continue;           // in the same state
    end if
  end do;
end for;
```

Fig. 7. Dynamic measurement process description

5 Test and Evaluation

Because we use the whole control flow graph as the program's expected behavior model, a tool that can analyze the binary code of executable will help significantly. There are a large number of related works in the areas of instrumentation of executables with the instrumentation consisting of both static and dynamic approaches. Most of the static analysis based intrusion detection systems are implemented by EEL[17], the Executable Editing Library which provides an abstract interface to parse and rewrite SPARC binary executables. But one of the limitations of EEL is that it is a tool within the SPARC platform and it is not supported by other platforms.

We use Pin [18][19], a tool for the dynamic instrumentation of programs to implement our model. Pin supports Linux binary executables; Windows executables; and MacOS executables on different platforms. With Pin's help, arbitrary code (written in C or C++) can be injected at arbitrary places in the executable. Pin adds the code dynamically while the executable is running. But at the same time, it is also possible to use pin to examine binaries such as images without running them. This is useful when we need to know static properties of an image. So Pin has the combined benefit of both static and dynamic analysis of the executables.

In our static analysis component, we use the Pin's static ability to examine the binaries in the image granularity, and the measurement monitor component is implemented with the help of Pin's dynamic instrument ability. Pin provides a

number of API calls in different granularity. Pin can instrument executables at instruction level, the basic block level, the routine level and the image level. So Pin can help the users to inject their own code when an image is loaded. We use Pin's API to make up our own pin tools. The IMG_AddInstrumentFunction API call can examine every image loaded by the program. We add our own static analysis arithmetic into the API and make up our own static analysis pin tool. Every time when an image is loaded, the code in the IMG_AddInstrumentFunction API function will be executed. The INS_AddInstrumentFunction API call can inject arbitrary code before the instruction is executed. And every time when an instruction is executed, the code in the IMG_AddInstrumentFunction API function will be executed. We add our own monitor code into this API call and make up our own dynamic monitor Pin tool. In addition, Pin provides a rich API that operates the executables. More details of other Pin API functions can be found in the Pin user's manual.

5.1 Test and Analysis

A code segment which overflow vulnerability exists is shown in Fig.8. Function pass () applies for 4 bytes of stack space to store the password that the user entered. If the password the user entered is identical with that the program sets, the function will print a welcome message. However, overflow vulnerability exists in this code segment. If the length of the password which is entered is 4 bytes, the program will run correctly. However, if the length is more than 4 bytes, the return address of the function will be overwritten. As a result, the function can't return normally. If the password which is entered is carefully designed, the program will jump to any location specified by the user.

```
/* overflow.c*/
#include <stdio.h>
int pass()
{
    char password[4];
    scanf("%s",password);
    if(strcmp(password,"1234")==0)
        printf("Welcome!\n");
           return 0;
}
int main()
{
    pass();
    printf("go ahead!\n");
           return 0;
}
```

Fig. 8. Return address overflow example

Our experiment of the behavioral-based dynamic measurement model is implemented using the open-source binary instrument tool PIN. The implementation includes three stages :firstly, static analyze the executable using a two-step scanning algorithm ,secondly , generate the expected behavior according the result of the first stage, and finally do the trust measurement to the program by checking whether the actual behavior of the running program corresponding with the expected behavior. The measurement process of the test code above is shown in Fig.9.

The first scan to the executable will generate all possible states the program may reach when it is running. For the test code segment above, a total of 23045 states may reach. After the second scan the pointer field *p1* and *p2* of every state node will be assigned a value. For the program's last state, its instruction's first address is 809f948. As this is the address of the last function, which indicates it is the last state, the analysis results show that the next state's address can't be found. During the dynamic measurement stage, a password is entered with more than 4 bytes. Then the returned address of the function pass() is overwritten, so measurement results shows that there has error in the return address 8048200 and the program is terminated.

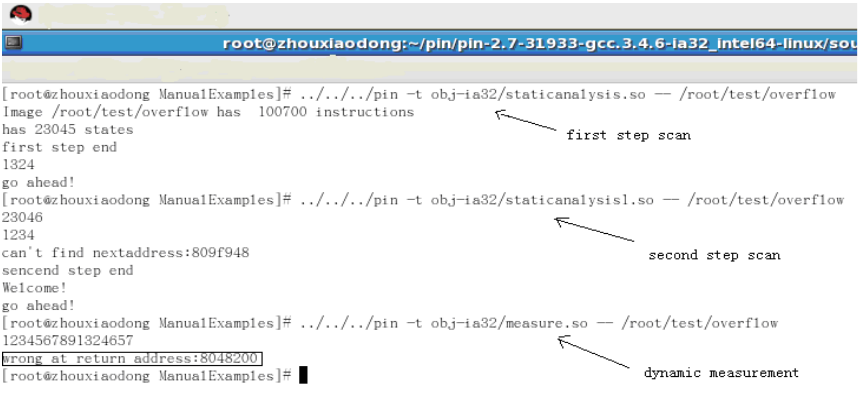

Fig. 9. Some test examples

5.2 Security Analysis

Measurement model presented in this paper integrates the static measurement of an executable program before it is loaded and dynamic measurement in the process of running, which increases security and reflects dynamic feature.

As for the malicious attacks which will modify the code(such as rootkits), because the integrity of the program files is destroyed , the program isn't allowed to run before passing the static measurement, which is the same as the traditional static measurement method, and it makes the program can't get the opportunities to execute. At the same time, this model can prevent some code injection attacks dynamically.

Buffer overflow attack is a typical code injection attack. If an intruder finds out a buffer overflow vulnerability of the program, the executing program may deviates from the expected way by inputting malicious parameters without modifying the

program file, and turns to execute other codes (usually the codes appointed by intruders). If intruders successfully attack the program, the control flow of the program will change dynamically. Here, the monitor will find that the actual control flow does not match the expected one, and the application will be terminated. Therefore, this model can prevent this kind of buffer overflow attack. Meanwhile, the model monitors the behavior of running program, which means that the model is dynamic compared with traditional static measurement method.

Because Pin runs in the user space, it may be possibly bypassed. However, when run as root, Pin is no more vulnerable than the kernel. And if all user programs in the system will under the protection of our model, the act of attempting to bypass Pin will also be observed. So in our opinion, it is feasible to implement the prototype with Pin.

6 Conclusions and Future Works

In this paper, we propose a dynamic measurement model to measure the executable's trust. If each program execution state is followed by its expected states based on the control flow analysis, that program can be said to be trusted. So the key problem is how to describe the expected behavior of one program. In our model, we take the control flow as the program's expected behavior. The control flow graph is analyzed in advance. And the result is the expected behavior of the program. If an executable is suffered from some code injection attacks, its control flow will be unexpected and that the program will be untrusted dynamically when it is running.

In our future works, we will further enhance the practicality of the expected behavior model. So far, our experiments are all based the static linked executables. Next, we will improve the analysis ability of dynamically linked executables by taking advantage of the Pin's dynamic instrument capability. At the same time, we will add the integrity measurement component to the implementation of the model to make the model more complete.

Acknowledgment. This work was supported by a grant from the Major State Basic Research Development Program of China (973 Program) (No.2007CB311106), the funding project for academic human resources development in institutions of higher learning under the jurisdiction of Beijing municipality.

References

1. Shen, C., Zhang, H., Wang, H., et al.: Survey of information security. Sci. China Ser. F-Inf. Sci. 50, 139–166 (2010) (in Chinese)
2. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A Secure and Reliable Bootstrap Architecture. In: IEEE Computer Society Conference on Security and Privacy (1997)
3. Arbaugh, W.A., Keromytis, A.D., Farber, D.J., Smith, J.M.: Automated recovery in a secure bootstrap process. In: Proceedings of Symposium on Network and Distributed Systems Security (NDSS), pp. 65–71 (1997)
4. Maruyama, H., Munetoh, S., Yoshihama, S., Ebringer, T.: Trusted platform on demand. IPSJ SIG. Notes Computer Security Abstract No.024-032

5. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium, pp. 9–13 (2004)
6. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-reduced integrity measurement architecture. In: SACMAT 2006: Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies (2006)
7. Shi, E., Perrig, A., van Doorn, L.: Bind: A Fine-grained Attestation Service for Secure Distributed Systems. In: Proceedings for the IEEE Symposium on Security and Privacy (2005)
8. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for Unix processes. In: IEEE Symposium on Security and Privacy, California, pp. 120–128 (1996)
9. Hofmeyr, S., Somayaji, A., Forrest, S.: Intrusion detection system using sequences of system calls. Journal of Computer Security 6(3), 151–180 (1998)
10. Chinchani, R., van den Berg, E.: A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 284–308. Springer, Heidelberg (2006)
11. Systrace (2011), http://en.wikipedia.org/wiki/Systrace
12. Intergrated Application Protection approaches such as Promon's Shield (2011), http://www.promon.no/integrated.html
13. Wespi, A., Dacier, M., Debar, H.: Intrusion Detection Using Variable-length Audit Trail Patterns. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 110–129. Springer, Heidelberg (2000)
14. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE Symposium on Security and Privacy, Oakland, California (2001)
15. Giffin, J., Jha, S., Miller, B.: Detecting manipulated remote call streams. In: 11th USENIX Security Symposium, San Francisco, California (2002)
16. Feng, H.H., Giffin, J.T., Huang, Y., Jha, S., Lee, W., Miller, B.P.: Formalizing sensitivity in static analysis for intrusion detection. In: IEEE Symposium on Security and Privacy, pp. 194–208. IEEE Press (2004)
17. Larus, J.R., Schnarr, E.: EEL: Machine Independent Executable Editing. In: SIGPLAN 1995 Conference on Programming Language Design and Implementation (1995)
18. Luk, C.-K., Cohn, R., Muth, R., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 12-15 (2005)
19. (2011), http://www.pintool.org/