Efficient Taint Analysis with Taint Behavior Summary

Ruoyu Zhang, Shan Huang, Zhengwei Qi
School of Software
Shanghai Jiaotong University
Shanghai China

Abstract—Software security has drawn much attention recently. As an effective approach to detect software vulnerabilities and improve software security, dynamic taint analysis has been frequently researched in the last few years. In this paper, we implement a dynamic taint tracking system LTTS. In order to address the efficiency problem which exists in many of the current taint tracking systems, we also propose a taint behavior summary mechanism to optimize the system. According to our experiments, LTTS has achieved relative high efficiency compared to the existing techniques.

Keywords—taint analysis; dynamic binary analysis;

I. INTRODUCTION

In the recent decades, there has been growing concerns with software security which is threatened by the software vulnerabilities. By IETF, software vulnerability is defined as “a flaw or weakness in a system’s design, implementation, or operation and management that could be exploited to violate the system’s security policy.” For instance, attacker can exploit the buff overflow vulnerability by causing an overflow on the stack, and overwrite the content reserved by the system. The attacker is able to control the procedure by changing the return address using the stack overflow.

There have been many researches on the detection of the software vulnerabilities[1][2]. The approaches can be generally categorized as static methods or dynamic ones. In most cases, static analysis is applied on the source code, and the dynamic approach aims at the binary. Since the source code of many software is hard to acquire, dynamic analysis is widely used in the software vulnerability detection.

Dynamic taint analysis is an effective method in the dynamic analysis for software vulnerability detection. The approach is based on the concept that any variable which can be influenced by an outside user is a potential threat to the program and should be monitored. In the beginning, certain memory blocks and registers which load outside data are marked as the source of tainting. During the execution of the program, the taint analyzer traces the taint propagation by marking every memory address and register influenced by the tainted data.

There has been much effort on the dynamic taint analysis[3], most of which suffer from high overhead. Dytan[4] is a generic dynamic taint analysis framework, which provides a customizable mechanism for the taint tracing, and its overhead is 50 times. Panorama[5] and LIFT[6] are also presented for the same purpose and slow down the target program by 20 times and 3.6 times respectively. The overhead is an important parameter for the runtime method, and the defects in efficiency have greatly limited the usage of this technique.

According to the common observation, large part of the binary codes in software are loaded from system library such as kernel32.dll, USER32.dll and ntdll.dll. The behavior of these modules is predictable, so it’s not necessary to check every instruction in them. We propose an solution to optimize the taint analysis technique by summarizing the taint effects of the library functions and idioms. The experiment result shows great reduction of the runtime overhead.

In this paper, we implement a systematic dynamic binary analysis based approach to apply taint analysis for software vulnerability detection. Aiming at the defect of low efficiency in current techniques, we improve our system with taint behavior summary, which will be described in detail in the next section.

Our contributions include:

- **Implement a system for dynamic taint analysis**
  We implement LTTS, a dynamic taint tracing system for binary code. The system marks the taint source, monitors the propagation of the tainted data, and generates alarms when the tainted data are involved in suspicious behavior.

- **Present a mechanism to optimize the efficiency of taint analysis**
  To reduce the overhead of the taint tracing techniques, this paper proposes a method of taint behavior summary. According to our experiments, it significantly lowers the runtime overhead of the system.

II. SYSTEM DESIGN

In this section, the design of the whole system will be introduced in the first place. In addition, we also describe the details of our approach including the implementation of the taint analyzer and the mechanism of taint behavior summary.
A. System Overview

LTTS is built on the binary instrument platform DynamoRIO[7], a runtime code manipulation system that supports code transformations on target program. The platform provides efficient instrumentation of binary by using code caching technology. Thanks to the facilitation of DynamoRIO, LTTS realizes the taint analysis with high efficiency.

Figure 1 provides an overview of LTTS. The system monitors the binary code stream with the binary instrument platform DynamoRIO, when the input of the outside user has been loaded into the target program, the taint source recognizer will detect the tainted memory blocks and initialize the taint tracking procedure. During the taint tracking, the malicious behavior asserter checks the propagation of the tainted data, and generates the report.

The main part of the taint behavior summary is the function call analysis. The general conception of this technique is to predict the taint propagation without concrete taint tracking in identifiable functions. The function recognizer is implemented to detect the calls to the libraries, and then the function analyzer is able to provide the taint behavior information according to the taint summary database.

B. Taint Source

Our approach is customized on the definition of the taint source. By default, LTTS marks all data which is introduced to the target program by the outside users as untrustful one. For example, since the malicious users can exploit buffer overflow vulnerabilities by manipulating the input file for the software, our system marks the location which loads the user file as a source of taint. When a file is read to the buffer, the memory blocks of the buffer are registered in LTTS, and monitored by the system.

C. Taint Tracking

After a tainted data is identified, the taint analysis engine is initialized to monitor each binary instruction which refers to the tainted data in order to track the behavior of the taint propagation. In order to determine the effect of the instructions, LTTS categorizes them into two sorts: (1) Instructions which can propagate the tainted data (LOAD, POP, PUSH etc.), (2) Instructions which do not affect the propagation of the tainted data(NOP, JMP, etc.).

It’s worthy of noticing that there are special cases used as idioms whose functions do not depend on the inputs. For example, “XOR EAX, EAX” always sets EAX to be zero regardless of whether the original value in eax is tainted. LTTS recognizes these idioms, and makes the taint tracking more efficient.

D. Malicious Behavior Assertion

Since the behavior of the tainted data is monitored, it is achievable for LTTS to detect the exploit on the software vulnerability. When processing the instructions, our instruction analyzer applies runtime analysis on it. In our approach, the system checks whether the tainted data is used as a control transfer target, such as a jump target, return address, or function pointer. According to our observation, many attacks attempt to overwrite one of these in order to interfere with the control flow of the victim program. In fact, the tainted data is very rarely used as a control transfer target in normal program, and the developers are not likely to handle the control of their codes into data which comes from the user[4].

E. Mechanism of Taint Behavior Summary

In the previous part of this section, we provide the approach of the baseline system. The system also suffers from high runtime overhead. In order to address this problem, we propose an approach to improve the efficiency by summarize the taint behavior of certain part of the code, and determine the taint propagation without processing every instruction. Our experiments show that this mechanism greatly reduces the overhead of LTTS.

API summary is a key part of the approach. Because most of the APIs in the system library can be determined without actually analyzing in every instance, we exam the code of
Figure 2. The formulas of the taint summary algorithm. TR stands for the taint relation in the target function, which contains taint vectors in the form of \(<Dst,Src,>\). TS is the set of the tainted data in the system scale, which records all the locations which are tainted, and should be monitored.

\[
\begin{align*}
TR_0 & = \Phi \rightarrow TS' = TS \quad (1) \\
TR_0 & \neq \Phi \land (TS) \land TR_0 \rightarrow TS' = TS - Tsrc \quad (2) \\
TR_0 & \neq \Phi \land (TS) \land TR_0 \rightarrow TS' = TS \cup \{ Tsrc \} \quad (3)
\end{align*}
\]

III. EVALUATION

Because this paper has proposed a mechanism to optimize the existing taint tracking techniques, the evaluation on the efficiency of LTTS is provided. We also apply the system on the software vulnerability detection to evaluate its effectiveness.

A. Efficiency

This subsection provides the performance of LTTS and the optimizations are described in the last section. The experiments is based on the SPEC CINT2006 benchmarks on Windows. The results of the experiments have shown that the taint behavior summary mechanism has greatly improved the efficiency of the system.

Figure 3 shows normalized execution time (the ratio of our time to native execution time) of LTTS. Because of the facilitation of the binary instrument platform DynamoRIO which also dynamically optimizes the execution of the target program, LTTS achieve the low overhead of 3.14 times on average without the taint behavior summary. When the summary is introduced to the system, the time cost reduces approximately by half, and the average runtime overhead is 1.41 times that of the native execution.

It can be observed from figure 3 that the performance of LTTS differs among the target programs. One reason is the structural distinction between them. The size of the exe module of the program is an important factor in the performance of our system. Since the library functions can be summarized and the analyzing time spent on them is relatively low. It’s also worth noticing that we haven’t summarized the system library entirely. When the program calls to the functions which haven’t been included in our taint behavior database, the overhead will increase.

B. Effectiveness

As a dynamic approach, LTTS is implemented at runtime to detect the existing threaten to the target system. We applied our approach on 7 real-world word processors(Kingsoft wps, justsystem ichitaro and hangul HWP are the word text processors for Chinese, Korean and Japanese respectively).

The target programs are running under the monitor of LTTS. The results are presented in Table I. In general, 7 exploitable vulnerabilities have been found, and another 57 ones maybe threatens.

IV. RELATED WORK

Binary analysis is an intuitive and effective way in program analysis, especially in the cases when the source code of the target programs are not acquirable.

In order to monitor the behavior of the running program for analysis, several tools are provided based on binary instrument. We build our tools on the base of DynamoRIO[7]. The platform presents a mechanism which dynamically removes much of the interpreted overhead from language implementations and has achieved high efficiency. There are also several other tools which can facilitate the dynamic binary analysis. Pin[8] and Valgrind[9] also provide great facilitation on this field of application.

Dynamic taint analysis is very common in runtime bug detection. Panorama[5] monitors the behavior of the target code, and it’s designed to detect integer bugs in malware. Minos[10] proposes hardware extension to perform runtime data flow integrity check to detect attacks. The authors...
perform checks at the whole-system level by modifying hardwares and the OS. In contrast, LTTS doesn’t require modifications of hardwares or the OS. Furthermore, the techniques described in this paper can be applied in many types of attack detections which are not addressed in Minos, such as format string overflow and security-sensitive variables overwrite.

V. Conclusion

In this paper, we present a taint tracking system LTTS, and propose a mechanism of taint behavior summary to optimize the system in efficiency. The experiments have shown that our approach greatly reduces the runtime overhead of the dynamic taint analysis. In the future, we will apply symbolic execution in the field of dynamic taint analysis, and broaden the application of the system.

References


