Static Vulnerabilities Detection Based on Extended Vulnerability State Machine Model

Bin Liang
School of Information, Renmin University of China, Beijing 100872, China
Key Laboratory of Data Engineering and Knowledge Engineering, MOE, Beijing 100872, China
liangb@ruc.edu.cn

Abstract—A static vulnerability detection method based on an extended vulnerability state machine is proposed in this paper. In this method, the state space of state machine model is extended. The security state of a variable can be identified by a property set that may consist of multiple security-related properties rather than a single property. As results, fine-grained state transition is provided to support accurate recognition of program security-related behaviors. Specially, the recognition of validation checking is introduced to reduce false positives. Besides, a systematic discrimination mechanism for preventing false negatives result from neglecting tainted data sources. The experimental results of a prototype system show that this method can effectively detect vulnerabilities in software systems with obviously lower false positive than existing methods, and avoid some serious false negative.

Keywords—vulnerabilities detection; static analysis; state machine

I. INTRODUCTION

In recent years, researchers have applied static analysis techniques to security vulnerabilities detection [1]. Taint-based analysis is the mainstream static detection approach. Based on the approach, static detectors traverse the control flows of target program and apply program variables state transition according to some vulnerability state machines. These state machines are defined to indentify the state transition of security-relative data based on specific vulnerability knowledge. As results, the detector can trace the tainted data propagation; when a tainted data is referenced at some security sink points, a possible vulnerability will be reported. Some static vulnerability detection systems adopt above approach have been developed and applied to complex software systems[2]. However, all of them are reported to have comparatively high false positive and false negative rates. The false positive rate of some detection systems is higher than 90% [3]. Users must refine detection results by manual code audit. Even worse, some serious vulnerabilities may be neglected due to false negative. One main reason is lack of accurate and effective identification and analysis of security-related program elements in existing methods and systems, e.g. data validation checking, tainted data source, etc.

To address the problem, we propose a static detection method based on an extended state machine model. In this method, we extend the state space of state machine model and employ fine-grained state description mechanism. The security state of a variable is identified by a property set that may consist of multiple security-related properties rather than a single property. So, fine-grained state transition can be provided to support accurate recognition of program security-related behaviors in parallel. Specially, the recognition of validation checking is introduced in state machine to reduce false positives. Besides, the concept of trusted perimeter is introduced to construct a systematic discrimination mechanism for preventing false negatives result from neglecting tainted data sources. The experimental results of a prototype system show that this method can effectively detect buffer overflow and other types of vulnerabilities in software systems with obviously lower false positive than existing methods and avoid some serious false negative.

II. MODEL AND METHOD

A. Detection Model

In our model, the security state space is extended for tracing multiple security properties evolving in parallel accurately. A set consist of multiple security-related properties is employed to describe the current security state of data. The basic elements of model are defined as below:

**Definition 1.** The model consists of the following components.
- \( \text{SRP} \): the set of security-related properties. SRP consists of various data properties related to security vulnerability, such as the source of data, validation checking passed, assignment state, etc.
- \( \text{SRO} \): the set of security-related operations. SRO consist of various operations which induce the data security-related properties being changed, such as memory copy, assignment, data comparison, etc.

**Definition 2.** A vulnerability state machine \( VM \) is described as a five-tuple \( \langle S, \Sigma, f, s_0, Z \rangle \), where:
- \( S = \text{P} (\text{SRP}) \), the set of security state, is a power set of \( \text{SRP} \).
- \( \Sigma = \text{SRO} \), control alphabet, consists of all security-related operations.
- \( f : S \times \Sigma \rightarrow S \), state transition functions, indentify the post state after enforcing a security-related operation at current state.
- \( s_0 = \emptyset \): initial security state.
- \( Z = \emptyset \): the set of final states, is a empty set in VM.
In a VM, the security state of a program variable is described with multiple elements of SRP set. The setting of SRP set is decided by specific target system and the type of vulnerability. In general, SRP will consist of three main parts as below.

1) Used to indentify whether a data variable is a trusted data, namely, under the control of malicious users.

A program bug maybe is not vulnerability. Only when the behaviors of the program function contains a security-related bug can be affected by the malformed data under the control of malicious users, the bug maybe result in actual security problem. Otherwise, the bug only is a potential program weakness rather than an exploitable vulnerability. So, the identification of the trustworthiness of data is the fundament of vulnerability detection.

2) Used to indentify what validation checking stages the tainted data have passed.

In general, system often performs some validations checking to the outer tainted data before referencing them. For example, operating system kernel should perform an upper-bound checking to a memory operation length data come form user space before a memory operation. In practice, validation checking has different implement pattern and structure, even for the validation checking with same function. Besides, validation checking may consist of multiple stages. Fox example, a validation checking to a length variable may be divided to three stages: sign checking, upper-bound checking, and lower-bound checking.

3) Used to indentify what security-sensitive operations have been applied to variable.

Some types of vulnerability are involved in the sequence of operations applied to data variables, e.g. double free, null pointer reference, etc. Accordingly, some elements corresponding to security-sensitive operations are included in SRP.

In this model, we can employ multiple security-related to describe the current security state of data rather than secure or not secure. Although the extension to security state space is comparatively simple, this setting makes it possible to accurately tracing multiple security properties evolving in parallel. It is especially important to validation checking.

B. Validation Checking Recognition

In traditional model, the validation checking is recognized as a whole, it requires static analysis is performed to comparative large program unit. However, a validation checking may consist of multiple stages that can be organized in random orders. It is very difficult or even impossible to generalize various possible implement pattern and structure. For example, an upper-bound checking may consist of checking of whether the length variable is a negative value and whether it is less than a buffer length. The order of the two checking stages is optional. In the detection model of MC [4], an upper-bound checking is regarded as a whole analysis unit. Unfortunately, the corresponding recognition rule can’t accommodate various implement patterns. It may result in serious false negative, e.g. there is a vulnerability existing in Linux Bluetooth device driver that is related with upper-bound checking but not found by MC [5].

The accurate recognition of validation checking can be achieved with fine-grain recognition process. As shown in Figure 1, a validation checking can be divided to multiple stages; SRP will include corresponding security-related properties to identify whether the target variable pass certain stage. A stage will correspond to a small program unit, e.g. one or two statements, that analysis engine can recognize it within an individual analysis step. Compared to recognition of validation checking as a whole, smaller recognition unit can provide more accurate results and more possibility. More importantly, the order of stages is not important to recognition. Because the recognition results of different stages are not overlap, we don’t need to predefine the order of validation checking stages. The recognitions of validation checking with same function but different implement pattern can be generalized to a static analysis rule. In practice programming, it is very commonly that the validation checking with same function may have many different implement patterns.

![Figure 1. Validation checking stages](image)

For a certain vulnerability type, we need to conclude all sorts of validation checking modes about it according to target system design and implement, and then refine them to a series of validation primitives and configure corresponding security states.

Validation checking has two main types. First, comparison and logical operations, such as equal, less than, greater than, etc., they are used to validate the upper/lower bound or the current value of a tainted data (may involved in multiple comparisons). Second, calling validation routines, they are some validation functions built into target systems for specific validation related to system logic.

For example, the VM related to an upper-bound validation to integer variables is configured as shown in Figure 2. For an unsigned tainted variable, if it is compared with a value and not greater than the value, its' security state will be directly pass to \{tainted, unsigned upper-bound checked\}, depicted as edge 3. But for a signed tainted variable, according to the type of data used to compare with the variable, the state transition will be determined respectively as follow:

- If it is not greater than a constant value, the post state will be \{tainted, signed upper-bound checked\}, depicted as edge 2.
If it is not greater than a signed value, the post state will be \{\textit{tainted, signed upper-bound checked}\}, depicted as edge 3.

After above two transitions, if the variable is not greater than an unsigned value or not less than zero, the state will be passed to \{\textit{tainted, unsigned upper-bound checked}\}, depicted as edge 4.

If it is not greater than an unsigned value, the post state will be \{\textit{tainted, unsigned upper-bound checked}\}, depicted as edge 5.

Besides, if a signed integer with a non-negative value, it can be regarded as an unsigned data, depicted as edge 6; after a checking similar to edge 1, state will be passed to \{\textit{tainted, signed upper-bound checked}\}, depicted as edge 7.

As shown in Figure 3, after determining the trusted perimeter and entry points of target system, entry points would be regarded as tainted data source.

![Figure 2. Integer upper-bound checking](image)

**C. Tainted Data Identification**

The initial state of a variable is an empty set $\emptyset$. The subsequent states are determined by the certain security-related operations applied to the variable. In general, analysis engine only concern the propagation of tainted data. The identification of the trustworthiness is closely related to the architecture and data process mechanism of target system. Based on the general architecture of software system, two definitions are introduced for tainted data identification as follow.

**Definition 3.** $TP$ (Trusted Perimeter) is a boundary between external space and trusted internal space, the data generated within $TP$ don’t subject to external operations.

**Definition 4.** Entry Point is a channel that introduces external data into $TP$.

The most important purpose of introducing $TP$ concept is tainted data identification. It is direct and natural to conclude a set of $TP$ entry points from the $TP$ setting of specific target system. The set not only includes the service interface of system, but also other all possible external data input channel, e.g. the network datagram receive routines of network stack in OS kernel.

As shown in Figure 3, after determining the trusted perimeter and entry points of target system, entry points would be regarded as tainted data source.

![Figure 3. Integer upper-bound checking](image)

For a set of entry point $EP$, it should be a subset of $SRO$ of target system, namely $EP \subseteq SRO$. For a variable $v$, corresponding VM state transition function $f_v$:

$$
\forall f_s \in EP, f_v(s, ep) = \text{\{tainted\}}
$$

In $f_v$, $s \in S$ is the current security state of $v$; $tainted \in SRO$ is the security state used to indentify tainted data.

Beside tainted data source, security state transition among variables should be considered; especially tainted data propagation. We need to consider two transition ways: assignment transition and memory copy transition.

The fundamental services of modern software, especially system software, are generally integrated in an isolated core with some interfaces for external service. For example, the OS kernel provides the fundamental functions for computer source management; the generation and operation of data within kernel aren’t impacted by user space programs. This general architecture feature of software system can be used to determine trusted perimeter and entry points.

**III. PROTOTYPE SYSTEM**

Based on above model and method, a prototype detection system is implemented. It can be employed to detect vulnerability in C source code.

![Figure 4. System architecture](image)
is not anticipant, it means there is a possible vulnerability; engine will output the vulnerability context information.

Using the prototype system, we perform detection experiments to some components of Linux kernel and MySql. Several potential vulnerabilities are found in experiments. The accurate rate of detection is 31.8% (22 total, 15 false positive, and 7 real). Especially, some vulnerabilities missed by traditional methods are located. Typically, a buffer overflow vulnerability that was neglected by MC [4] and Coverity [2] is located in Linux kernel file systems code as shown below.

```c
static void fill_psinfo(struct elf_prpsinfo *psinfo,
    struct task_struct *p, struct mm_struct *mm)
{
    int i, len;
...
    1227 len = mm->arg_end - mm->arg_start;
    1228 if (len >= ELF_PRARGSZ)
        1229 len = mm->arg_end-1;
    1230 copy_from_user(psinfo->pr_psargs,
            (const char __user *)mm->arg_start, len);
...
    1419 fill_psinfo(psinfo, current->group_leader, current->mm);
```

In above code segment, although there is an upper-bound checking for tainted len variable before referencing it in a copy_from_user calling at line 1228, a negative value (when mm->arg_end is less than mm->arg_start) can bypass the checking. As a result, when calling copy_from_user, redundant data will be copy to a kernel buffer from user space and causing a buffer overflow at line 1230.

When detecting, copy_from_user calling is a security sink point; a check point is set at it. The corresponding anticipated security state of len parameter is $\emptyset$ or \{tainted, unsigned upper-bound checked\}. In above code, mm parameter is stem from current task structure data (line 1419) that can be control by malicious users via loading a malformed binary file. The macro current is regard as an entry point of TP. The content of mm will be indentified as tainted data. Consequently, the security state of len will be set to \{tainted\} at line 1227. If the result of judgment at line 1228 is false, the security state will be set to \{tainted, signed upper-bound checked\}. Finally, the actual security state is different with anticipate states, a report will be sent by detection engine.

To the vulnerability, the main reasons of false negative in existing methods are the macro current not to be regarded as a tainted data source and the non-effective validation checking not to be indentified.

Besides, Linux kernel fixed the vulnerability with changing the type of len variable to unsigned integer. The prototype system can recognize the change and don’t produce false positive for fixed code.

IV. RELATED WORKS

Engler et al. develop several correlative static detection tools MC, XGCC, etc. [4] [6]. These tools are evolved to a bossiness product Coverity. It has found hundreds of vulnerabilities and bugs in Linux kernel. In contrast, our prototype system provides more accurate vulnerability property recognition rather than more vulnerability types. Based on an extended vulnerability state machine model, the prototype system can found some neglected vulnerability from Linux kernel code checked by Coverity. Johnson et al. use type qualifier technique to detect user/kernel pointer vulnerability in Linux kernel [3]. They extended the type inference ability of Cqual tool to support context sensitive analysis. Shankar et al. use Cqual to detect format string vulnerability [7]. The main problem of these methods is lack of the recognition for validation checking and produce over many false positives and false negatives.

V. CONCLUSION

In this paper, a static vulnerability detection method based extended vulnerability state machine is proposed. We extent the state space of state machine model; the security state of a variable is identified by a property set that may consist of multiple security-related properties rather than a single property. Based on the model, fine-granularity state transition is provided to support accurate recognition of program security-related behaviors; the recognition of validation checking is introduced in vulnerability state machine to reduce false positives. A prototype system is implemented based on compiler technique. The experiment shows that this method can effectively detect vulnerabilities in system software with obviously lower false positive than existing static detection methods and avoid some serious false negative. In the future, we will apply the method to more vulnerability type and other programming languages, such as java, php, and asp, etc.

ACKNOWLEDGMENT

This work has been supported by the National Natural Science Foundation of China under Grant No.60703102, No. 60873213 and No.60703103; the Beijing Science Foundation under Grant No.4082018; the National 863 High-tech Program of China under Grant No. 2007AA01Z414; the Open Project of MOE Key Laboratory of Data Engineering and Knowledge Engineering.

REFERENCES