LETTER

Systematic Detection of State Variable Corruptions in Discrete Event System Specification Based Simulation

Hae Young LEE†, Member and Jin Myoung KIM††a), Nonmember

SUMMARY In this letter, we propose a more secure modeling and simulation approach that can systematically detect state variable corruptions caused by buffer overflows in simulation models. Using our approach, developers may not consider secure coding practices related to the corruptions. We have implemented a prototype of the approach based on a modeling and simulation formalism and an open source simulator. Through optimization, the prototype could show better performance, compared to the original simulator, and detect state variable corruptions.

key words: data corruption, state variable corruption, buffer overflow, secure coding, discrete event system specification

1. Introduction

When simulation models are implemented in C/C++, e.g., using adevs[1], they are usually implemented as classes. During a simulation run, most of their instances, including their state variables, created with operator new, are allocated in the heap [1], [2]. Like other computer programs, these state variables in the heap could be corrupted by buffer overflow (BOF) vulnerabilities [3] that would be probably caused by developers making mistakes [4]. In consideration of applications of simulation, state variable corruptions (SVCs) would lead to reaching erroneous decisions, so that SVCs must be detected [3].

However, detecting or preventing their corruptions has not been well explored since most techniques for detecting or preventing data corruptions have focused on control data (e.g., on the stack) [5]. For example, AddressSanitizer [6] of Clang can detect heap based BOFs that cause control flow hijacking. However, it does not protect adjacent buffers in classes from overflow [7]. We may force developers to comply with secure code practices, or to use security extensions, e.g., YARRA [8]. But it would be much better to create developer-centric approaches for secure modeling and simulation [4].

This letter presents a secure discrete event system simulator, called SeaDevs, that can detect SVCs caused by BOFs in Discrete Event System Specification (DEVS) [9] models written in C/C++, by extending Lee’s previous work [3]. SeaDevs uses canaries for detecting BOFs that may cause SVCs, and systematically checks through its abstract simulator. The current implementation of SeaDevs is based on adevs. In order to check the occurrence of BOFs, it automatically places canaries and checks them during simulation loops. Therefore, developers may leave secure coding practices related to BOFs that may cause SVCs out of consideration. Through optimization, we could achieve some performance enhancement, compared to adevs, as well as the capability to detect SVCs. While the use of canaries is neither new nor complex, we deliver a developer-friendly and practical solution, based on understanding of the data structure and logic of C/C++ DEVS simulators.

2. Problem Statement and DEVS Abstract Simulator

This section describes the problem that we address in this letter and a brief overview of the DEVS abstract simulator.

2.1 Problem Statement

Basically, this letter addresses SVCs caused by BOFs in DEVS model instances. SVCs may be caused by many reasons, including BOFs [3], accesses to freed memory [10], and hardware faults [11]. But BOFs are a particularly important type of memory corruptions since they are often exploited by adversaries [10]. For example, users or external processes may inject malformed inputs into vulnerable model instances to exploit BOFs [3].

Figure 4 shows an example of a vulnerable DEVS model class in which BOF may corrupt constant proc_t (i.e., BOF causes SVC), the processing time of the model. Detecting BOF in the example is not easy since BOF does not change flow control and not occur privilege escalation. Also, BOF may not cause SVC. However, it must be detected since the vulnerability would be still exploitable. Therefore, the letter focuses on detecting BOFs that may cause SVCs.

2.2 DEVS Abstract Simulator

Simulation of a hierarchical DEVS model is performed by

```cpp
class B: public Atomic <>
{
    // ...
    char msg1 [4];
    char msg2 [4];
    int proc_t;
};
```

Fig. 1 Example of a vulnerable model class.
atomic simulators associated with the terminal models of the hierarchical model, coordinators associated with digraph models in the hierarchical model, and a root coordinator [9]. The overall simulation loop is implemented by the root-coordinator. The root-coordinator sends a message to its direct subordinate, corresponding to the all-inclusive digraph model, and then receives the time of the next event within the digraph model. The procedure is repeated until some termination condition is met.

3. Abstract Simulator for SeaDevs

SeaDevs employs canaries to detect SVCs of atomic DEVS model instances. If BOF occurs within an atomic DEVS model instance, some state variables of the model instance may be corrupted (i.e., SVCs may occur). But it would also corrupt canaries between the buffer and the corrupted variables in the instance. Thus, SVCs can be detected by checking the canaries.

Thanks to the sound simulation framework of DEVS, canaries can be systematically checked through the abstract simulator for SeaDevs. As shown in Fig. 2, our abstract simulator uses an additional type of message, c-message, to check all canaries in a hierarchical DEVS model. The root-coordinator sends c-message to its direct subordinate whenever canaries need to be checked (e.g., at the end of a simulation run). The message is then sent to all atomic simulators (blue solid lines), probably via coordinators. Upon receiving the message, an atomic simulator checks canaries, if any, in the associated terminal model (a green dotted line), and then sends the result to its direct superior, corresponding to the digraph model that has the associated terminal model. Finally, the root-coordinator controls – continues or terminates – the simulation loop based on the results (red dotted lines).

Although the use of canaries to detect SVCs was first proposed by Lee’s approach [3], SeaDevs has drastically enhanced usability for developers as well as security. In Lee’s approach, developers should find all buffers and place preprocessors for canaries at the end of each of the buffers. Also, they should locate all check points where the canaries need to be checked at and place preprocessors for the check points at the places. If they have missed some buffers or checkpoints, the associated models may be vulnerable to SVCs. In contrast, SeaDevs provides automatic insertions of canaries and check points (see Sect. 4.2).

Canaries in Lee’s approach are checked, which involves some overhead, whenever functions having checkpoints are called. In contrast, SeaDevs enables canaries to be checked systematically with the abstract simulator. That is, the time of checking canaries can be centrally controlled. By reducing canary check overhead, the performance of SeaDevs has improved, compared to that of adves (see Sect. 5.1).

4. Implementation of SeaDevs

This section describes our implementation of SeaDevs in details.

4.1 Class Template for DEVS Models

Like adves, a custom atomic model class in SeaDevs is also implemented by deriving from class template Atomic which is a subclass of class template Devs. Our template Devs for SeaDevs has an additional pure virtual member function, checkCanaries, which is used to check canaries. The function of a custom atomic model class is automatically implemented by the preprocessor of SeaDevs, as described in Sect. 4.2.

4.2 Preprocessor

Before compiling source codes for a hierarchical DEVS model class, the preprocessor of SeaDevs scans the source code for buffers. On detecting a buffer, the preprocessor automatically places a canary code, a 32-bit variable whose value is 0, right after the buffer. Figure 3 shows an example of canary codes placed by the preprocessors, in a custom atomic model class.

Then, the preprocessor places an implementation of checkCanaries for each custom atomic model class. The function returns 0 if BOF is not occur in the instance. Since all canaries in the class are initially set to be 0 as shown in Fig. 3, an occurrence of BOF would make some of the canaries nonzero. That is, the function would return a nonzero value in case of BOF. Figure 4 shows an example of the function that the preprocessor automatically places in the

```java
class B: public Atomic
{
    //...
    char msg1[4]; int FRINGE_CANARY_1 = 0;
    char msg2[4]; int FRINGE_CANARY_2 = 0;
    int proc_t;
};
```

Fig. 2 Abstract simulator for SeaDevs.

Fig. 3 Automatic insertion of canaries.
4.3 Simulator Class

In adevs, the overall simulation loop is implemented by an instance of class Simulator. Class Simulator for SeaDevs has an additional member variable that determines the interval between c-message in simulation time. The default value is infinity, which makes all canaries checked at the end of a simulation run. Developers can freely adjust the interval to make canaries checked more frequently.

5. Performance Evaluation and Limitations

In this section, evaluation results are given to show the effectiveness of SeaDevs. Limitations of the current implementation are also discussed.

5.1 Performance Evaluation

To show the effectiveness of SeaDevs in terms of SVC detection and performance, we have compared SeaDevs with adevs and Lee’s approach [3]. Our evaluation results are shown in Table 1. The performance enhancement could be achieved through some optimization (see Appendix). In addition to the performance enhancement, SeaDevs could achieve the detection of SVCs. Moreover, developers using SeaDevs may not consider canaries and canary check points in source code, while developers using Lee’s approach should carefully place preprocessor directives for canaries and check points in source code; any missing directive in Lee’s approach may cause SVCs.

We have evaluated the performance of adevs that was compiled with Clang and AddressSanitizer [6]. Developers using AddressSanitizer of Clang may not consider secure coding. However, AddressSanitizer could not detect SVCs since the SVCs did not hijack control-flow. Even if AddressSanitizer may provide the detection of SVCs someday, the overhead would be too heavy to run complex simulations.

5.2 Limitations and Discussions

Although the current implementation of SeaDevs could detect SVCs caused by BOFs, it has some limitations: First, it cannot detect BOFs that occur with filling zeros, e.g., using function memcpy. Second, simulation loops may be interrupted due to BOFs that do not cause SVCs (i.e., false positives). Lastly, it cannot detect SVCs caused by other reasons, such as hardware faults [11].

The use of canaries for detecting BOFs [12]–[15] is a well-known technique. For example, StackGuard [12] uses canaries to detect stack BOFs, as shown in Fig. 5(a), and has been deployed on mainstream operating systems. However, it cannot detect neither stack BOFs that only affect local variables nor heap based BOFs [15]. ValueGuard [13] enables stack BOFs that only affect local variables to be detected, by inserting canaries in front of all variables, as shown in Fig. 5(b). Canaries are used to detect heap BOFs that overwrite blocks’ boundaries in ValueGuard (Fig. 5(c)) and HeapTherapy [14] (Fig. 5(d)). However, both ValueGuard and HeapTherapy they are incapable of detecting heap based BOFs that corrupt data inside a single memory block, shown in Fig. 5(e). This is a limitation of most existing defense mechanisms since they are difficult to change the layout of memory blocks that many developers rely on [13]. In contrast, SeaDevs can detect such heap based BOFs, by inserting canaries right after each buffer in an atomic model class, as shown in Fig. 5(f).

In existing mechanisms, canaries are checked only when blocks are freed (HeapTherapy) or whenever blocks are accessed (ValueGuard). Some mechanisms, such as PointGuard [16], use encryption to protect pointers that address memory blocks but would involve relatively high overhead. In contrast, SeaDevs can detect SVCs more efficiently, by using the abstract simulator that can control the time of checking canaries, with the consideration of simulation progress.

Although SeaDevs currently addresses SVCs due to

```cpp
class B: public Atomic {
{
//...
virtual int checkCanaries () const {
    return (FRINGE_CANARY_0|FRINGE_CANARY_1|0);
}

Fig. 4 Automatic implementation for checking canaries.
```
BOFs in C/C++ DEVS simulators, the idea of SeaDevs may be applied to other C/C++ formalism-based simulators (e.g., a Petri net simulator [17]) and formal methods. By understanding the data structure of such programs, we may be able to detect BOFs and SVCs that cannot be detected by existing mechanisms. By understanding the logic of the programs, we may be able to do that more efficiently.

6. Conclusion and Future Work

In this letter, we presented SeaDevs that can detect SVCs caused by BOFs. Thanks to the systematical framework of DEVS, BOFs that may cause SVCs can be automatically detected through the abstract simulator. The current implementation of SeaDevs is developer-friendly, so that developers may not consider SVCs caused by BOFs. Thus, SeaDevs, compared to Lee’s approach [3], could drastically enhance usability for developers as well as security, although it is based on the approach. The performance of SeaDevs was shown with the evaluation results.

We will improve SeaDevs so that current limitations described in Sect. 5.2 could be removed or relaxed. Also, we will investigate other security problems in simulation as described in Sect. 5.2 could be removed or relaxed. Also, we will try to apply the idea of SeaDevs to other formalism-based simulators.

References

[1] Adevs: A Discrete Event System simulator, https://web.ornl.gov/~nutaroj/adevs/
[2] T.G. Kim and S.B. Park, “The DEVS formalism: Hierarchical modular systems specification in C++,” Proc. 1992 European Simulation Multiconf., June 1992.
[3] H.Y. Lee, “Buffer overflow detection in DEVS simulation using canaries,” Proc. 2017 Winter Simulation Conf., pp.4554–4555, Dec. 2017.
[4] M. Green and M. Smith, “Developers are Not the Enemy!: The Need for Usable Security APIs,” IEEE Security & Privacy, vol.14, no.5, pp.40–46, Oct. 2016.
[5] S.A. Carr and M. Payer, “DataShield: Configurable Data Confidentiality and Integrity,” Proc. 2017 ACM Asia Conf. on Comput. and Commun. Security, pp.193–204, April 2017.
[6] AddressSanitizer – Clang 10 Documentation, https://clang.llvm.org/docs/AddressSanitizer.html
[7] E. Wimberley, “Bypassing AddressSanitizer,” https://dl.packetstormsecurity.net/papers/general/BreakingAddressSanitizer.pdf, Sept. 2013.
[8] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn, “Modular Protections against Non-control Data Attacks,” Proc. 24th IEEE Comput. Security Foundation Symp., pp.131–145, June 2011.
[9] B.P. Zeigler, T.G. Kim, and H. Praehofer, Theory of Modeling and Simulation, 2nd Ed., Academic Press, 2000.
[10] F. Qin, S. Lu, and Y. Zhou, “SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs,” Proc. 11th Int’l Symp. High-Performance Comput. Arch., pp.291–302, Feb. 2005.
[11] B. Fang, P. Wu, Q. Guan, N. DeBardeleben, L. Monroe, S. Blanchard, Z. Chen, K. Pattabiraman, and M. Ripeanu, “SDC is in the Eye of the Beholder: A Survey and Preliminary Study,” Proc. 46th Annual IEEE/IFIP Int’l Conf. Dependable Syst. and Netw. Workshop, pp.72–76, June 2016.
[12] C. Cowan et al., “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks,” Proc. 7th USENIX Security Symp., Jan. 1998.
[13] S.V. Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens, “ValueGuard: Protection of native applications against data-only buffer overflows,” Proc. 6th Int’l Conf. on Infor. Syst. Security, vol.6503, pp.156–170, Dec. 2010.
[14] Q. Zeng, M. Zhao, and P. Liu, “HeapTherapy: An Efficient End-to-end against Heap Buffer Overflows,” Proc. 45th Annual IEEE/IFIP Int’l Conf. Dependable Syst. and Netw., pp.485–496, June 2015.
[15] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, “To Detect Stack Buffer Overflow with Polymorphic Canaries,” Proc. 48th Annual IEEE/IFIP Int’l Conf. on Dependable Syst. and Netw., pp.243–254, June 2018.
[16] C. Cowan et al., “PointGuardTM: Protecting pointers from buffer overflow vulnerabilities,” Proc. 12th USENIX Security Symp., Aug. 2003.
[17] M. Drzazga, P. Blasiak, and D. Dylowicz, petri-net-simulator, https://github.com/bprzemyslaw/petri-net-simulator

Appendix

In the current implementation of adves, a simulation run is carried out with member function execUntil of class Simulator. In the function, another member function nextEventTime is called twice. Each call takes time since checking all the atomic model is involved within the call. Thus, we have optimized the function as Fig. A.1.

---

**Fig. A.1** Optimization of adves.