Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

BUFFER OVERFLOW DETECTION IN DEVS SIMULATION USING CANARIES

Hae Young Lee

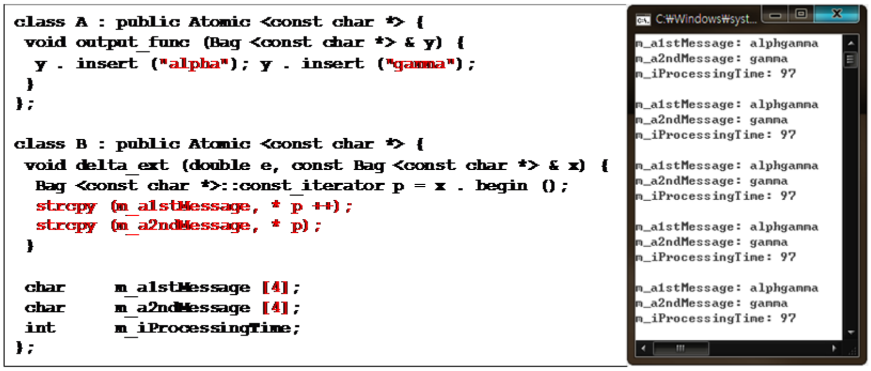
DuDu IT
96 Gamasan-ro
Seoul, 08501, SOUTH KOREA

ABSTRACT

This paper addresses buffer overflows (BOFs) in simulations written in the C/C++ language, which could be exploited by attackers to pollute simulation results. The paper then presents a BOF detection method for Discrete Event System Specification, in which canaries placed after buffers are used to detect BOFs. A concept-of-proof of the proposed method that uses a custom preprocessor has been implemented and shows that BOFs can be detected with minimal modifications.

1 INTRODUCTION

Buffer overflows (BOFs) could exist within simulations developed in the C/C++ programming language (Lee 2017). Figure 1 shows an example of BOFs in a simulation developed using *adevs* (Nutaro 2017), which is a C++ library for M&S based on Discrete Event System Specification (DEVS) (Zeigler, Praehofer, and Kim 2000). In the example, the processing time, while it should be a constant, of an instance of `class B` was initially 100, but is later corrupted to 97 due to BOFs triggered by a `class A`'s instance. Malformed inputs that cause such BOFs could be injected by other instances of models, users through user interfaces (UIs), other processes through inter-process communications (IPC), and so on. These BOFs would not be easy to be detected since they may not result in memory access errors or simulation crashes. However, they could, due to 'pollution' of the simulation results, lead to reaching erroneous decisions in consideration of applications of simulations. Thus, BOFs must be detected before resulting in 'the pollution.'



```
class A : public Atomic <const char *> {
 void output_func (Bag <const char *> & y) {
  y . insert ("alpha"); y . insert ("gamma");
 }
};

class B : public Atomic <const char *> {
 void delta_ext (double e, const Bag <const char *> & x) {
  Bag <const char *>::const_iterator p = x . begin ();
  strcpy (m_a1stMessage, * p ++);
  strcpy (m_a2ndMessage, * p);
 }

 char     m_a1stMessage [4];
 char     m_a2ndMessage [4];
 int      m_iProcessingTime;
};
```

```
m_a1stMessage: alphgamma
m_a2ndMessage: gamma
m_iProcessingTime: 97

m_a1stMessage: alphgamma
m_a2ndMessage: gamma
m_iProcessingTime: 97

m_a1stMessage: alphgamma
m_a2ndMessage: gamma
m_iProcessingTime: 97

m_a1stMessage: alphgamma
m_a2ndMessage: gamma
m_iProcessingTime: 97

m_a1stMessage: alphgamma
m_a2ndMessage: gamma
m_iProcessingTime: 97
```

Figure 1: Example of buffer overflows in simulations.

This paper proposes a canary based method for detecting BOFs within C/C++ based DEVS simulations. An integer variable called 'canary' is first inserted right after every buffer in atomic DEVS models. Since the state of an atomic model may change in the external and internal transition functions, the canar-

978-1-5386-3428-8/17/$31.00 ©2017 IEEE

ies are then checked right after each execution of these functions in order to detect BOFs. A proof-of-concept of the method for *adevs*, in which statements for BOF detection using canaries are placed by a custom preprocessor, shows that BOFs in *adevs* models can be easily detected.

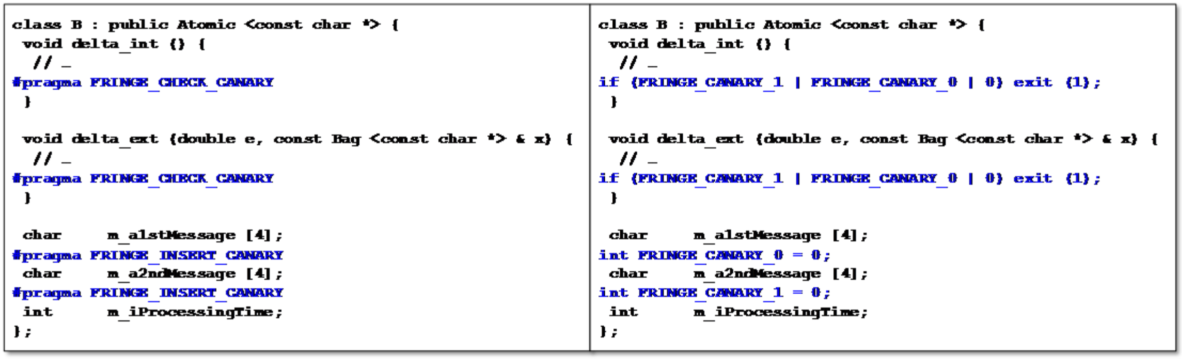
2 BOF DETECTION FOR DEVS USING CANARIES

Attackers could exploit BOFs in simulations to lead simulation runs to produce erroneous results. To this end, they could inject malformed inputs into 'vulnerable' models through UIs, IPCs, files, and so on. To detect such BOFs, canaries are placed and then checked in the proposed method. For each atomic DEVS model, a canary, which is initially set to 0, is placed right after every buffer in the model. While writing data to a buffer, the program may overrun the buffer's boundary. That is, a BOF occurs, resulting in overwriting the adjacent canary with a non-zero value. But by checking the canaries before returning to the caller, the BOF can be detected and reported to the user.

In an atomic DEVS model, member variables, including buffers, are corresponding to the state set, and thus allowed to change in the external and internal transition functions. That is, BOF could occur only within these functions. Thus, by checking the values of canaries right after each execution of the functions, BOFs can be detected; a BOF has occurred if a canary having a non-zero value exists. If we need not consider BOFs triggered internally, i.e., we assume all BOFs are externally triggered, canaries need to be checked only after the execution of the external function.

3 IMPLEMENTATION

A proof-of-concept has been implemented for *adevs*, in which a custom preprocessor replaces preprocessor directives with statements for BOF detection as shown in Figure 2. This procedure could be automated by customizing a compiler.



```
class B : public Atomic <const char *> {
 void delta_int () {
  // …
#pragma FRINGE_CHECK_CANARY
 }

 void delta_ext (double e, const Bag <const char *> & x) {
  // …
#pragma FRINGE_CHECK_CANARY
 }

 char     m_a1stMessage [4];
#pragma FRINGE_INSERT_CANARY
 char     m_a2ndMessage [4];
#pragma FRINGE_INSERT_CANARY
 int      m_iProcessingTime;
};
```

```
class B : public Atomic <const char *> {
 void delta_int () {
  // …
if (FRINGE_CANARY_1 | FRINGE_CANARY_0 | 0) exit (1);
 }

 void delta_ext (double e, const Bag <const char *> & x) {
  // …
if (FRINGE_CANARY_1 | FRINGE_CANARY_0 | 0) exit (1);
 }

 char     m_a1stMessage [4];
int FRINGE_CANARY_0 = 0;
 char     m_a2ndMessage [4];
int FRINGE_CANARY_1 = 0;
 int      m_iProcessingTime;
};
```

Figure 2: Preprocessing for BOF detection in the method.

4 CONCLUSION AND FUTURE WORK

This paper presented a BOF detection method for DEVS. BOF could exist in not only C/C++ based DEVS simulations but also any other scientific programs, and be serious in consideration of their applications. This will be further investigated.

REFERENCES

Lee, H.Y. 2017. "Buffer Overflow Vulnerabilities in DEVS Models". In *Proc. of the 2017 KSS Spring Conf.* (available at: https://sites.google.com/site/whichmeans/publications/2017-KSS-Spring-Paper.pdf).

Nutaro, J. 2017. *adevs*. http://web.ornl.gov/~nutarojj/.

Zeigler, B.P., Praehofer, H., and Kim, T.G. 2000. *Theory of Modeling and Simulation*, 2nd Ed.