

# Method of integer overflow detection to avoid buffer overflow

Zhang Shirui<sup>1</sup> Xu Lei<sup>2</sup> Xu Baowen<sup>2</sup>

(<sup>1</sup>School of Computer Science and Engineering, Southeast University, Nanjing 211189, China)

(<sup>2</sup>State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

**Abstract:** A simplified integer overflow detection method based on path relaxation is described for avoiding buffer overflow triggered by integer overflow. When the integer overflow refers to the size of the buffer allocated dynamically, this kind of integer overflow is most likely to trigger buffer overflow. Based on this discovery, through lightly static program analysis, the solution traces the key variables referring to the size of a buffer allocated dynamically and it maintains the upper bound and lower bound of these variables. After the constraint information of these traced variables is inserted into the original program, this method tests the program with test cases through path relaxation, which means that it not only reports the errors revealed by the current runtime value of traced variables contained in the test case, but it also examines the errors possibly occurring under the same execution path with all the possible values of the traced variables. The effectiveness of this method is demonstrated in a case study. Compared with the traditional buffer overflow detection methods, this method reduces the burden of detection and improves efficiency.

**Key words:** integer overflow; buffer overflow; path relaxation

Integer overflow has become a key factor it triggering buffer overflow<sup>[1]</sup>. The research from CERT/CC indicates that, compared with all kinds of security exploits of the C/C++ program, buffer overflow accounts for more than 50%; and majority of these buffer overflow exploits are triggered by integer overflow, such as the SSH CRC32 remote overflow, the apache chunked-encoding overflow, and the OPENSSH challenge-response overflow, etc.

Researchers have provided multiple solutions for preventing and detecting buffer overflow, such as static code inspection and dynamic detection based on testing; however, most of these methods ignore the importance of the integer overflow which is a key factor causing buffer overflow. The relationship between integer overflow and buffer overflow has been largely underestimated and needs to be carefully studied.

This paper mainly focuses on a particular kind of integer overflow which can trigger buffer overflow and proposes a simplified integer overflow detection method based on path relaxation. This method first identifies variables which need to be traced by lightly static program analysis, then it inserts a code for maintaining constrained information and finally detects the integer overflow that will actually or potentially occur by dynamic testing. Path relaxation indicates that it not

only reports the integer overflow revealed by the current runtime data, but also examines the integer overflow possibly occurring under the same execution path with different inputs. The simplification indicates that it reduces the range of variables that need to be traced by merely maintaining the constraint information for the variables only related to the size of the allocated buffer dynamically, such as the parameters of operator new, function malloc() and calloc().

## 1 Integer Overflow

The standard of ISO C99 indicates that the integer overflow will lead to “uncertain results”, but it does not define what compilers must do. Therefore, most of the compilers (such as Visual C++ 6.0) choose to ignore the integer overflow and the consequence is that an unexpected value will be stored<sup>[2]</sup>. This unexpected value is usually smaller than the actual one because it is the result of the expected integer value mod the upper bound of the integer variable with the pre-condition that the expected value is greater than the upper bound. When the integer overflow refers to the size of the dynamically allocated buffer, this kind of integer overflow is most likely to trigger the buffer overflow. This paper focuses on the typical situation in a C/C++ program where potential buffer overflows triggered by integer overflows exist.

As mentioned above, if the integer overflow refers to the size of the dynamically allocated buffer, it makes the actual allocated size smaller than the expected size. An instance of the integer overflow is shown as follows:

```
int myfunction(int * array, int len)
{
    int * myarray, i;
    myarray = malloc(len * sizeof(int));
    if (myarray == NULL)
    {
        return -1;
    }
    for(i = 0; i < len; i++)
    {
        myarray[i] = array[i];
    }
    return myarray;
}
```

where the allocated size of array “myarray” is determined by the product of parameter “len” and the size of variable type int. If this product overflows, as the wrong product stored is smaller than the expected one due to the integer overflow, it will make the actual memory size allocated for “myarray” smaller than the expected size which should contain the “len” number of int variables. When “myarray” is initialized by using the value of “len” as its elements’ number, it is ex-

Received 2009-01-07.

**Biographies:** Zhang Shirui (1984—), male, graduate; Xu Baowen (corresponding author), male, doctor, professor, bwxu@nju.edu.cn.

**Foundation items:** The National Natural Science Foundation of China (No. 60873050, 60703086), the Opening Foundation of State Key Laboratory of Software Engineering in Wuhan University (No. SKLSE20080717).

**Citation:** Zhang Shirui, Xu Lei, Xu Baowen. Method of integer overflow detection to avoid buffer overflow [J]. Journal of Southeast University (English Edition), 2009, 25(2): 219 – 223.

tremely possible that data is written into unallocated space, which results in buffer overflows.

The traditional buffer overflow detection methods usually ignore the importance of integer overflows which can indirectly trigger buffer overflows. There are generally two traditional detection methods: the static overflow detection method and the dynamic overflow detection method. The static detection method is basically a kind of code audit rather than the execution of the program<sup>[3]</sup>. The code is examined by static analysis to detect buffer overflows. The dynamic detection method is basically a testing method executed by running well-designed test cases and analyzing the executed results<sup>[4]</sup>. There are widely used detection tools for both static and dynamic methods. The typical static detection tools include: Splint, CSSV<sup>[5]</sup>, ESC; while the typical dynamic detection tools include: STOBO<sup>[6]</sup>, Lhee and Chapin<sup>[7]</sup>, Purify<sup>[8]</sup>.

In addition to these traditional methods, Horowitz from the McAfee Company proposed a method of preventing buffer overflow by detecting the integer overflow<sup>[9]</sup>. His method indicates that in a program where the potential buffer overflow triggered by the integer overflow exists, there must be at least one loop with a huge number as its upper limit. So the strategy is to locate all these kinds of loops and check them one by one. However, as the standard of the huge upper limit is set manually by users, the effort of this method is closely related with users' experience.

## 2 Integer Overflow Detection Based on Path Relaxation

In order to prevent buffer overflow by detecting the integer overflow, this paper proposes a simplified integer overflow detection system based on path relaxation.

### 2.1 Path relaxation

Traditional dynamic detection methods indicate that the single test case can only detect the single execution path and the variables in one execution path can correspond to different inputs. An instance of a small program with a single execution path is shown as follows:

```
int * myarray;
int len;
myarray = malloc( len * sizeof( int ) );
```

The program only has a single execution path. The test cases of the program can easily cover all the possible execution paths. There is no guarantee that it is always safe after these test cases have been passed, because the variables in the same path can be specified with different values.

As mentioned above, even if one program has a single execution path, the traditional dynamic detection methods cannot detect all the potential errors with limited test cases. Therefore, path relaxation is introduced. Instead of searching more execution paths for testing, path relaxation just focuses on one execution path and relaxes the values of the variables in the path. In other words, it maintains constraints for variables, which means building the upper bound and the lower bound of the variables in one execution path. With the help of path relaxation, the detection method not only depends on the current runtime data of variables in test cases, but also

depends on the constraints of these variables. Through path relaxation, the detection can give warnings for potential errors.

### 2.2 Simplifying tracing variables

As discussed in section 1, the integer overflow referring to dynamic buffer allocation is most likely to trigger buffer overflow. Compared with traditional static analysis, our method reduces the range of tracing variables, since only variables closely related to the dynamic buffer allocation are traced.

The particular types of tracing variables in our method include: the parameter of operator new, the parameters of functions malloc( ) and calloc( ). These variables need to be traced because they determine the actual sizes allocated for buffers, which will indirectly result in buffer overflow if the integer overflow occurs.

### 2.3 Building detection system

For a given program, the detection system is generated by applying the simplified integer overflow detection method based on path relaxation. First, the tracing variables are identified by the principles discussed in section 2.2. Then each tracing variable is combined with constraint information: the upper bound and the lower bound of variable values. For each integer variable, the initial constraint is demonstrated as  $[INT_{min}, INT_{max}]$ , where the  $INT_{min}$  stands for the minimum value of an integer variable and  $INT_{max}$  stands for the maximum value of an integer variable. The constraints of tracing variables will change according to the constraint changing rule, which will be discussed in the next section. After the constraint information is inserted into the program, the part for the static analysis is over and the part for dynamic testing starts. When the program within constraint information is tested and executed to the points related to dynamic buffer allocation, the detection system detects integer overflows that actually or potentially occur according to the examining rule, which also will be discussed in the next section. The construction of the constraint changing rule and the examining rule is vital in this integer overflow detection system.

## 3 Rules

When the program is executed, the constraint changing rule defines how the upper bound and the lower bound of each tracing integer variable change when the program is executed to such points referring to dynamic buffer allocation. The examining rule defines how to detect the actual or potential integer overflow by both the current value of variables and the constraints of variables. The contents of each rule are shown in Tab. 1 and Tab. 2.

In Tab. 1,  $x, y, z$  stand for two types of tracing variables: the parameters of new, malloc( ) and calloc( ) and the variables on which these parameters have data dependency.  $L(x)$  stands for the lower bound of tracing integer variable  $x$  while  $U(x)$  stands for the upper bound of tracing integer variable  $x$ , and it is the same for  $L(y), U(y), L(z)$  and  $U(z)$ . Path constant  $c$  is actually not a real constant, since it stands for the variables whose values will not change under the current paths. Only the three most popular arithmetic operations are

**Tab. 1** Constraint changing rule

Id	Statement	Constraint changing rule
1	$x = y$	$L(x) = L(y), U(x) = U(y)$
2	$x < y$	$L(x) = L(x), U(x) = \min(U(x), U(y)); L(y) = \max(L(x), L(y)), U(y) = U(y)$
3	$x ! = y$	$L(x) = L(x), U(x) = U(x)$
4	$x = c$	$L(x) = U(x) = c$
5	$x < c$	$L(x) = L(x), U(x) = \min\{c, U(x)\}$
6	$x > c$	$U(x) = U(x), L(x) = \max\{c, L(x)\}$
7	$x! = c$	If $L(x) = c$ , then $L(x) = c + 1, U(x) = U(x)$ ; if $U(x) = c$ , then $U(x) = c - 1, L(x) = L(x)$ ; else $L(x) = L(x), U(x) = U(x)$
8	$x = y + c$	If $c \geq 0$ and $U(y) + c \leq \text{INT}_{\max}$ , then $L(x) = L(y) + c, U(x) = U(y) + c$ If $c \geq 0$ and $U(y) + c > \text{INT}_{\max}$ , then $L(x) = L(y) + c, U(x) = \text{INT}_{\max}$ , give warning: potential integer overflow If $c < 0$ and $L(y) + c \geq \text{INT}_{\min}$ , then $L(x) = L(y) + c, U(x) = U(y) + c$ If $c < 0$ and $L(y) + c < \text{INT}_{\min}$ , then $L(x) = \text{INT}_{\min}, U(x) = U(y) + c$ , give warning: potential integer overflow
9	$x = y * c$	If $c \geq 0$ and $U(y) * c \leq \text{INT}_{\max}$ , then $L(x) = L(y) * c, U(x) = U(y) * c$ If $c \geq 0$ and $U(y) * c > \text{INT}_{\max}$ , then $L(x) = L(y) * c, U(x) = \text{INT}_{\max}$ , give warning: potential integer overflow If $c < 0$ and $L(y) * c \geq \text{INT}_{\min}$ , then $L(x) = L(y) * c, U(x) = U(y) * c$ If $c < 0$ and $L(y) * c < \text{INT}_{\min}$ , then $L(x) = \text{INT}_{\min}, U(x) = U(y) * c$ , give warning: potential integer overflow
10	$x = y + z$	If $U(y) + U(z) \leq \text{INT}_{\max}$ , then $L(x) = L(x) + L(y), U(x) = U(x) + U(y)$ If $U(y) + U(z) > \text{INT}_{\max}$ , then $L(x) = L(x) + L(y), U(x) = \text{INT}_{\max}$ , give warning: potential integer overflow
11	$x = y * z$	If $U(y) * U(z) \leq \text{INT}_{\max}$ , then $L(x) = L(x) * L(y), U(x) = U(x) * U(y)$ If $U(y) * U(z) > \text{INT}_{\max}$ , then $L(x) = L(x) * L(y), U(x) = \text{INT}_{\max}$ , give warning: potential integer overflow
12	$x = y \% z$	$L(x) = 0, U(x) = \max( L(z) ,  U(z) ) - 1$

**Tab. 2** Examining rule

Id	Statement	Examining rule
1	<code>malloc(num + sizeof(header))</code>	If $T(\text{num}) + \text{sizeof}(\text{header}) > \text{INT}_{\max}$ then give error report: integer overflow occurs If $U(\text{num}) + \text{sizeof}(\text{header}) > \text{INT}_{\max}$ then give warning: potential integer overflow
2	<code>malloc(num * sizeof(Object))</code>	If $T(\text{num}) * \text{sizeof}(\text{Object}) > \text{INT}_{\max}$ then give error report: integer overflow occurs If $U(\text{num}) * \text{sizeof}(\text{Object}) > \text{INT}_{\max}$ then give warning: potential integer overflow
3	<code>calloc(num, sizeof(Object))</code>	If $T(\text{num}) * \text{sizeof}(\text{Object}) > \text{INT}_{\max}$ then give error report: integer overflow occurs If $U(\text{num}) * \text{sizeof}(\text{Object}) > \text{INT}_{\max}$ then give warning: potential integer overflow
4	<code>new Object[num]</code>	If $T(\text{num}) * \text{sizeof}(\text{Object}) > \text{INT}_{\max}$ then give error report: integer overflow occurs If $U(\text{num}) * \text{sizeof}(\text{Object}) > \text{INT}_{\max}$ then give warning: potential integer overflow

included in the constraint changing rule, they are addition, multiplication and mod, each of which corresponds to one type of data dependency. The warnings given in the constraint changing rule indicate that the integer overflow might occur at that particular point.

In Tab. 2, num stands for the parameters of new, malloc() and calloc() which must be traced;  $T(\text{num})$  stands for the current value of num;  $U(\text{num})$  stands for the upper bound of num at the detecting point according to the constraint changing rule while  $L(\text{num})$  stands for the lower bound of num at the detecting point also based on the constraint changing rule. As shown in Tab. 2, there are two final reports for detection. One is the error report, which indicates that the integer overflow indeed occurs at the detecting point by checking  $T(\text{num})$ , the current value; the other is the warning report, which indicates that the integer overflow might occur at the detecting point by checking  $U(\text{num})$  and  $L(\text{num})$ .

#### 4 Case Study

To validate the effectiveness of our new method, the multimedia plug-ins from one famous open source multimedia framework, Gstreamer (<http://gstreamer.freedesktop.org/src/gst-plugins>), is chosen to be the source-code for testing. The potential integer overflow that can trigger the buffer overflow has been detected after applying this method in the source-code of plug-in 0.8.3. The part of the source-code which contains the detected integer overflow is shown as follows:

org/src/gst-plugins), is chosen to be the source-code for testing. The potential integer overflow that can trigger the buffer overflow has been detected after applying this method in the source-code of plug-in 0.8.3. The part of the source-code which contains the detected integer overflow is shown as follows:

```
gst_also_src_mmap(GstAlsa * this, snd_pcm_sframes_t
* avail)
{
    snd_pcm_uframes_t offset;
    snd_pcm_channel_area_t * dst;
    const snd_pcm_channel_area_t * src;
    int i, err, width = snd_pcm_format_physical_width(
this ->format ->format);
    GstAlsaSrc * alsa_src = GST_ALSA_SRC(this);
    /* areas points to the memory areas that belong to
gstreamer. */
    (a) dst = g_malloc(this ->format ->channels *
sizeof(snd_pcm_channel_area_t));
    if(((GstElement *) this) ->numpads == 1){
        /* interleaved */
    (b) for(i = 0; i < this ->format ->channels; i++){
```

```

dst[ i ]. addr = als_a_src - > buf[ 0 ] - > data;
dst[ i ]. first = i * width;
dst[ i ]. step = this - > format - > channels * width;
}
} else {
/* noninterleaved */
(c) for( i = 0; i < this - > format - > channels; i + + ) {
    dst[ i ]. addr = als_a_src - > buf[ i ] - > data;
    dst[ i ]. first = 0;
    dst[ i ]. step = width;
}
}
}

```

For the above given program above after applying our approach, a warning for the potential integer overflow is given at line (a). Our method first enforces a slight static analysis in order to identify the tracing variables. According to the principle of simplifying tracing variables in our method, only the parameter of `malloc()`: `this - > format - > channels` and those having data dependency on it are traced. After initial constraints are set for them, the boundary value is maintained by the constraint changing rule. When the program within constraint information is executed to detect line (a), detection is imposed according to the examining rule. There are two steps for detection:

**Step 1** Test whether the production between the current value of `this - > format - > channels` and `sizeof( snd_pcm_channel_area_t )` is greater than  $INT_{max}$ . The testing result is that the production is smaller than  $INT_{max}$ , then goes to the next step;

**Step 2** Test whether the production between `U( this - > format - > channels )` and `sizeof( snd_pcm_channel_area_t )` is greater than  $INT_{max}$ . The testing result is that this production exceeds the value of  $INT_{max}$ , then the method gives warnings indicating that the potential integer overflow will occur at detecting line (a).

Through the process of dynamic detection, it is found that the value for `sizeof( snd_pcm_channel_area_t )` is 0x8 and the value for `this - > format - > channels` is 0x1fffeeda at line (a), the product between them is

$$0x8 * 0x1fffeeda = ffff76d0$$

which is smaller than  $INT_{max}$ , 0xffffffff (based on a 32-bit computer). But the constraint value for `this - > format - > channels` at line (a) is [0x0, 0x20000004], so the upper bound is 0x20000004 and the production between the upper boundary value and `sizeof( snd_pcm_channel_area_t )` is

$$0x20000004 * 0x8 = 0x100000020$$

which is greater than  $INT_{max}$ , 0xffffffff. Therefore, it is possible that if the upper bound of the interval [0x0, 0x20000004] is assigned for `this - > format - > channels`, the integer overflow will occur and the actual size allocated for buffer `dst` is

$$0x100000020 \% 0xffffffff = 0x21$$

which is much smaller than the expected size for that buffer, 0x100000020.

Assuming that the integer overflow indeed occurs at line

(a), which leads to the real size allocated for `dst` is much smaller than the original size, when the program is executed to line (b), as the “for” loop corresponds to writing data into buffer `dst`, it results in writing data to unallocated space and leads to the buffer overflow; if the program is executed to line (c), the statement is still responsible for writing data into buffer `dst` and it also leads to the buffer overflow just as in line (b).

## 5 Conclusion and Future Work

This paper proposes a simplified integer overflow detection method based on path relaxation in order to detect dangerous integer overflows that will trigger buffer overflows. This new approach has two advantages: It can improve the efficiency of static analysis by simplifying tracing variables while reducing dependency on the value of variables in testing by path relaxation. In general, our method represents a totally different way when compared with traditional methods. Traditional methods focus on whether data are written into unallocated space, which is the direct cause of buffer overflow; our new approach focuses on whether the size of the buffer is allocated less at the beginning, which is an important indirect cause for buffer overflow. In the future, we will focus on other types of integer overflows because the integer overflow related to buffer allocation is just one kind. Although the integer overflow referring to the size of an allocated buffer is most likely to trigger buffer overflow, it is still possible that buffer overflow can be triggered by other kinds of integer overflows. Another task for future work is to address the performance impacts by adding an analysis phase to the compiler. This phase will make possible optimizations for the inserted code for maintaining constraint information of tracing variables.

## References

- [1] Blexim. Basic integer overflows [EB/OL]. (2002-12-28) [2008-06-08]. <http://www.phrack.com/issues.html?issue=60&id=10#article>.
- [2] Aho A, Sethi R, Ullman J. *Compilers: principles, techniques, and tools* [M]. New York: Addison-Wesley, 1986: 188 – 200.
- [3] Laroche D, Evans D. Statically detecting likely buffer overflow vulnerabilities[C]//*Proc of the 10th USENIX Security Symposium*. Washington, DC, 2001: 177 – 190.
- [4] Wilander J, Kamkar M. A comparison of publicly available tools for dynamic buffer overflow prevention[C]//*Proc of the 10th Network and Distributed System Security Symposium*. San Diego, 2003: 149 – 162.
- [5] Dor N, Rodeh M, Sagiv M. Towards a realistic tool for statically detecting all buffer overflows in C[C]//*Proc of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. San Diego, 2003: 155 – 167.
- [6] Haugh E, Bishop M. Testing C programs for buffer overflow vulnerabilities[C]//*Proc of the 10th Network and Distributed System Security Symposium*. San Diego, 2003: 123 – 130.
- [7] Lhee K, Chapin S. Type-assisted dynamic buffer overflow detection[C]//*Proc of the 11th USENIX Security Symposium*. San Francisco, 2002: 81 – 88.
- [8] Hastings R, Joyce B. Fast detection of memory leaks and access errors[C]//*Proc of the Winter USENIX Conference*.

San Francisco, 1992: 125 – 136.

[9] Horovitz O. Big integer loop protection[EB/OL]. (2002-12-

28) [2008-06-08]. <http://www.phrack.com/issues.html?issue=60&id=9#article>.

## 一种防止缓冲区溢出的整数溢出检测方法

张实睿<sup>1</sup> 许 蕾<sup>2</sup> 徐宝文<sup>2</sup>

(<sup>1</sup> 东南大学计算机科学与工程学院, 南京 211189)

(<sup>2</sup> 南京大学计算机软件新技术国家重点实验室, 南京 210093)

**摘要:** 为了防止由整数溢出引起的缓冲区溢出, 提出了一种简化的基于路径松弛的整数溢出检测方法. 表示动态分配缓冲区大小的整型变量发生溢出, 极有可能引发缓冲区溢出. 该检测方法基于这一发现, 在动态测试之前先进行轻量级的静态分析, 跟踪与动态分配缓冲区大小相关的关键变量, 保存追踪的关键变量在不同地方的取值上限和下限, 并将维护信息插入源代码中. 测试时通过路径松弛, 在执行路径上不仅考虑追踪变量的当前测试用例值, 判断程序是否出现整数溢出, 还根据插入的维护信息进一步考虑追踪变量可能的取值范围, 判断程序是否有可能出现整数溢出. 实例研究验证了该方法的有效性, 并且与同类方法相比, 减少了检测量, 提高了检测效率.

**关键词:** 整数溢出; 缓冲区溢出; 路径松弛

**中图分类号:** TP309.2