Computers & Security 89 (2020) 101616

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Static detection of real-world buffer overflow induced by loop

Peng Luo [a], Deqing Zou [a,*], Yajuan Du [b], Hai Jin [a], Changming Liu [a], Jinan Shen [a]

[a] Cluster and Grid Computing Lab, Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan 430074, China
[b] School of Computer Science and Technology, Wuhan University of Technology, Wuhan 430074, China

ARTICLE INFO

Article history:
Received 23 December 2017
Revised 13 November 2018
Accepted 10 December 2018
Available online 17 September 2019

Keywords:
Buffer overflow
Security vulnerability
Loop
Data movement
Static detection technique

ABSTRACT

Buffer overflow (BoF) is one of the most dangerous security vulnerabilities. A BoF can be induced by functions, such as the *memcpy* family, or loops with pointer or array operations. Static detection of BoF is a well-defined method and often performed before system deployment. However, most of previous static techniques either detect the BoFs induced by functions or analyze simple loops that induce BoFs, such as the single loop with a single loop variable. In order to clearly understand the impact of loops on BoF, we perform an empirical study on real-world BoFs detected in the past three years, including 211 BoFs from 60 open-source systems, and find that about a third of these BoFs are induced by loops. Motivated by this observation, we propose a novel static detection technique, DBloop, to localize BoFs induced by loops. The key of DBloop is to get the extremum length of data-movement on the buffer by analyzing target loops and then check whether the buffer overruns using constraint solving. We have implemented DBloop with CIL, and evaluated it on real-world programs detection by comparing with a commercial tool, Checkmarx, and an open-source tool, Splint. DBloop successfully localizes 57 BoFs induced by loops, while Checkmarx and Splint only detect 4 and 3, respectively. Moreover, DBloop has detected 4 new BoFs that have not been reported before.

© 2019 Published by Elsevier Ltd.

1. Introduction

C/C++ is one of the most popular language, and there are a lot of programs developed by C/C++ for different reasons, such as performance and historical legacy. However, the property of C/C++ language that allows developers directly and manually manipulate memory, often leads to programs vulnerable to memory errors. In C/C+ programs, mistakes in memory manipulation and misuses of APIs can easily induce security vulnerabilities. Buffer overflow (BoF) is one of the most serious and common vulnerability that occurs when data movement overruns the allocated bound of a buffer because of missing bound checking or input validation before memory manipulation or API calling. BoF can cause data corruption, system crash, denial of service, and arbitrary code execution.

The BoFs can be induced by functions or loops that operate on pointers and arrays. We call the BoFs induced by functions as fun-BoFs, and the BoFs induced by loops as loop-BoFs. Some fun-BoFs are induced by functions that contain loops in the internal code. In general, these functions do not cause BoFs, which occur only with special cases. The functions inducing BoFs include I/O functions, memory manipulation functions, string manipulation functions, and memory allocation functions that allocate insufficient memory. In the early 2017, Wikipedia wantonly spread in global world and performed ransomware attack to large numbers of users, by exploiting a fun-BoF in Windows. The loops can induce BoFs when the loops contain data movement operations with pointers or arrays which overrun the bounds of allocated buffers. The loop-BoFs are subtle and need to be seriously considered.

After decades of study, many BoF detection techniques have been proposed and can be generally divided into two types: dynamic testing (Haugh and Bishop, 2003; Lhee and Chapin, 2002; Wang et al., 2010; Xu et al., 2008; Zeng et al., 2015) and static program analysis (Khedker, 2014; Viega et al., 2000; Wagner et al., 2000; Wang et al., 2012; Yamaguchi et al., 2014; 2013). Some dynamic testing techniques identify BoFs by giving a large number of generated inputs, and others by inserting special code into programs or combining above two approaches. For instance, Ccured (Necula et al., 2002b) assumes boundary of variables should not be exceeded by all accesses and inserts checking code into programs to identify BoF occurrences. The advantages of dynamic testing techniques are that they rarely report false warnings because they have execution information (Fang and Hafiz, 2014; Ye et al., 2016). But, their disadvantages are that they have limited code coverage because it is impossible to traverse all paths of a large system. In general, the static program analysis techniques scan source

* Corresponding author.
E-mail addresses: deqingzou@hust.edu.cn (D. Zou), dyj@whut.edu.cn (Y. Du).

https://doi.org/10.1016/j.cose.2019.101616
0167-4048/© 2019 Published by Elsevier Ltd.

codes of programs to pinpoint the code segments with possible BoFs. The advantages of such schemes are the efficiency on scanning large systems. The static detection techniques can check the source code before program deployment and help developers eliminate BoFs, especially, from large systems.

The previous static BoF detection techniques apply different kinds of methods, including pattern/model analysis (Yamaguchi et al., 2013), information-flow analysis (Yamaguchi et al., 2014), and constrain solving (Wang et al., 2012). The pattern/model techniques localize suspicious operations that meet with special patterns. For instance, ITS4 (Viega et al., 2000) scans source code and reports warning for dangerous API functions, such as the *strcpy* family, while Chucky (Yamaguchi et al., 2013) exposes the operations missing checks. The techniques based on information-flow analyze the source code by applying data-flow or control-flow analysis, such as the popular commercial tool, Checkmarx. The constrain-solving techniques are efficient on checking the boundary of a buffer, and many of them apply symbolic execution, such as ARCHER (Xie et al., 2003). However, most of these techniques ignore the loop-BoFs by focusing on the dangerous API functions.

A few existing static techniques detect vulnerable loops or loop-BoFs. Rawat and Mounier (2012) argue that BoFs can also be induced by loops that contain data-movement operations, and present a static approach to identify these loops in binary executables. Parfait (Cifuentes and Scholz, 2008) only considers the array bound in single loop but cannot analyze the nested loops and that with paths. Li et al. (2010) present a constrain-solving method using symbolic execution to localize BoFs. Dowser (Haller et al., 2013) also adopts symbolic execution and takes a lot of time in the learning phase to obtain execution paths. However, these methods are inefficient for the complex loops which contain nested loops with multiple loop variables. Moreover, for large systems, the symbolic execution methods are time consuming and inefficient because of path explosion.

In this paper, we strive to detect loop-BoFs induced by different kinds of loops. To this end, we first perform an empirical study on real-world BoFs that have been detected in the past three years, which argues the importance of loop-BoFs detection and guides the process of loop-BoF detection. Motivated by the study results, we propose a novel static detection technique, DBloop, to localize loop-BoFs.

Our empirical study is based on the collection work of BoFs from the National Vulnerability Database (NVD). We collect the BoFs found from 2015 to 2017 indexed by NVD. We perform selection on the collected BoFs, and only keep the BoFs from open-source systems with vulnerable code segments. We get 211 BoFs from 60 open source systems,[1] such as Apache, PHP, LibreOffice, and TigerVNC. After that, we perform comprehensive study on the BoFs: examining the *vulnerability patterns*, *exploits* and *fix strategies*. For each BoF, we carefully examine its vulnerability report, corresponding source code, exploit method and related patches, all of which together provide us a relatively thorough understanding of the BoF characteristics. Our study reveals many meaningful findings as shown in Table 1, which provide useful guidelines for loop-BoF detection.

Based on the above study results, DBloop mainly analyzes loops and corresponding operations of data-movement with arrays and pointers to localize loop-BoFs. The key insight of DBloop is to first get the size of a buffer, such as the size of a array or allocated heap memory block, and then check whether data-movement of the targeted loop overruns the bound of the buffer. In detail, DBloop consists of four steps to localize the loop-BoFs:

[1] The BoFs is available at https://github.com/pluohust/BoF.

Table 1
Our findings of real world loop-BoF characteristics and implications for loop-BoF detection.

Findings	Implications
(1) There are 68 loop-BoFs, accounting for 32.2% of the total BoFs.	BoF detection tools also need to detect loop-BoFs which have been overlooked.
(2) The loop-BoFs are induced by the operations of data-movement with arrays or pointers.	Focusing on operation of data-movement with arrays and pointers is a good simplification for loop-BoF detection.
(3) The cause of BoFs with arrays is that the data-movement exceeds the bound of arrays.	Analyzing data-movement operations with arrays and the bound of corresponding arrays can expose the BoFs with arrays.
(4) The cause of BoFs with pointer is that the data-movement exceeds the bound of buffers in stack/heap.	Analyzing data-movement operations with pointers and the allocated size of corresponding buffers can expose the BoFs with pointers.

(1) Identify suspicious loops that contain operations of data-movement using arrays or pointers.
(2) Apply code-slicing on target loops, and classify target loops according to whether the loops are nested and whether the loops contain paths.
(3) Perform traceability analysis to obtain the bound of a buffer.
(4) Identify BoFs using constraint solving.

We have implemented DBloop with CIL (Necula et al., 2002a), and evaluated it by analyzing real-world applications and comparing with a commercial tool, Checkmarx, and an open-source tool, Splint (Evans and Larochelle, 2002). We use DBloop and these two tools to detect known loop-BoFs, the 68 loop-BoFs collected from 60 applications. DBloop has successfully localized 57 BoFs induced by loops, while Checkmarx and Splint only detected 4 and 3, respectively, which reveals the effectiveness of our approach. We also implement DBloop to analyze 20 popular open-source applications, and DBloop pinpoints 4 new BoFs that have not been reported before.

The main contributions to this paper are listed as follows:

- We perform an empirical study on real-world BoFs, to provide useful guidelines for BoF detection and avoidance.
- We present a static loop-BoF detection technique, DBloop, to localize loop-BoFs from the source code of programs using constraint solving.
- We implement DBloop with CIL and evaluate it by detecting BoFs from real-world applications. The results show that DBloop can effectively detect BoFs with a reasonable false positive rate.

The rest of the paper is organized as follows. Our empirical study is introduced in Section 2. Section 3 presents the analysis on loop bound, and our static loop-BoF detection technique, DBloop, is described in Section 4. Evaluation results are presented and analyzed in Section 5. Section 6 presents related work, and Section 7 concludes the paper.

2. Empirical study on BoF

To recognize the development of BoF in real-world software systems, we perform an empirical study on real-world BoFs that can guide us how to detect BoFs. We collect the BoFs found from 2015 to 2017 indexed by the National Vulnerability Database (NVD). Then we select part of BoFs on the collected dataset, and only keep the BoFs from open-source applications with vulnerable code segments. We also remove the BoFs that are caused by the same code segment in the same program. Table 2 shows the selection results, the detailed information of the BoFs. There are 211

Table 2
Subject systems.

Subjects	Description	#BoF bugs
OCaml	OCaml compiler	1
ADFlib C	Library for Amiga formatted devices	1
DBD::mysql	DBI driver for MySQL	1
GNU C Library	Core libraries for the GNU systems	2
Libarchive	Multi-format archive and compression library	2
Atheme	A customiz able IRC services daemon	1
ZooKeeper	API for open-source server	1
PHP	Scripting language for hypertext	3
QEMU	Machine emulator and virtualizer	4
OpenJPEG	Open-source JPEG 2000 codec	3
Wireshark	Network protocol analyser	19
cracklib	A library to enforce strong passwords	1
Firefox	Web browser	45
libcsp	A library for threads	1
Boa Webserver	Web server	1
BlueZ	Bluetooth protocol stack	1
ImageMagick	Images processing software	6
Libksba	A library for X.509 certificates and CMS data	1
LibVNCServer	C libraries for VNC server	2
LibTIFF	Libraries for the Tag Image File Format	9
cJSON	An JSON parser	1
VLC	Multimedia player and framework	1
MuPDF	Lightweight PDF viewer	1
Redis	In-memory data structure store	1
w3m	Text-only web browser	1
FFmpeg	Solution for multimedia data	26
Bro	Network analysis framework	2
exfat-utils	A package handling exFAT filesystem	1
Pillow	Friendly PIL	1
mod_cluster	httpd-based load balancer	1
libass	Subtitle renderer for the ASS/SSA	1
Libming	A library for generating Macromedia Flash files	3
Libdwarf	A library for reading (and writing) applications	1
libgit2	A library for Git core	1
libevent	An event notification library	1
JasPer	Program handling image data	2
GNU oSIP	An LGPL implementation of SIP	4
LibreOffice	Powerful office suite	2
GnuTLS	Implementation of the TLS, SSL and DTLS	2
MapServer	Open Source platform for mapping applications	1
TigerVNC	Client/server applications for VNC	1
zziplib	A library providing read access on ZIP-archives	1
Virgil	Research project creating virtual 3D with GPU	2
apngdis	Package deconstructing APNG file	1
radare2	Reversing framework	3
vim	Text editor	2
Tcpreplay	A suite of test tools	1
libimobiledevice	Protocol library for iOS devices	2
Capstone	Multi-architecture disassembly framework	1
kernel	Linux kernel	22
PCRE	Program handling pattern matching	2
Gnulib	GNU basic libraries	1
libsndfile	C library for reading and writing files	5
elfutils	Utilities and libraries handling ELF files	1
Google gRPC	RPC framework	1
fen	Image viewer	1
ImageWorsener	Utility and library for image processing	1
rzip	Compression program	1
Ettercap	Suite for man in the middle attacks	1
Underbit MAD	MPEG audio decoder	2

BoFs from 60 open-source systems. To follow representative principle, we choose one typical example for each BoF type in most systems. For systems that have more than one BoF type, we choose two or more BoF examples. There are four typical systems, including *Wireshark*, *Firefox*, *FFmpeg* and *Linux kernel*.

After the empirical study on the real-world BoFs, we find that the direct causes of BoFs are from two sources, *function* and *loop*, at the level of application source code. It is well-known that many library functions can cause BoFs without bound checking, such as the *memcpy* and *strcpy*. However, there are also many BoFs hidden in the loops that perform data movement. To go along with this, the BoFs are divided into fun-BoFs and loop-BoFs, respectively. Then, we introduce the study of these two kinds of BoFs.

2.1. Fun-BoFs

There are 143 fun-BoFs, accounting for 67.8% of the total BoFs. The functions inducing fun-BoFs are divided into three types: *memory-allocation*, *I/O* and *data-movement*, as shown in Fig. 1. The following sub-sections outline the three types of functions.

2.1.1. Memory-allocation

In our study, there are 29 BoFs induced by the memory-allocation functions, accounting for 20.2% of the BoFs induced by functions. According to statistical analysis, the memory-allocation functions inducing BoFs include *malloc*, *calloc*, *alloca*, *kmalloc*, *realloc*, etc.

We analyze the code segments of these BoFs, and conclude their trigger processes as follows:

1. Program calls memory-allocation library functions to allocate memory for a buffer, but it does not allocate enough size of memory.
2. Program writes data to the buffer and overruns the bound of the buffer.

Fig. 2 shows an example of BoF induced by *kmalloc* function. This BoF is in a service driver for Linux kernel. This program allocates a block of buffer, the size of which is *sizeof(struct apr_data) + user_payload_size* (line 204). Then the program would write data to the buffer (line 214). However, the integer variable *apr_request → payload_size* is determined by inputs, which may overflow. If it overflows, the allocated buffer will be smaller than the needed, and there is a buffer overflow which leads to denial of service. The BoF can be avoided by checking and restricting the length of data-movement. However, it also violates the developers' original intention, which may lead to data loss. Thus, this type of BoFs are induced by the allocation functions.

2.1.2. I/O

In our study, there are 23 BoFs induced by I/O functions, accounts for 16.1% of the total BoFs induced by functions, which is the least in these three types. According to statistical analysis, the I/O functions includes *sprintf*, *write*, *scanf*, *sscanf*, *read*, *snprintf*, etc.

After the analysis of code segments, these trigger processes are available:

1. Program obtains/outputs data by calling the I/O functions, but it imposes no or too weak restrictions for the data.
2. The input/output data overruns the bound of the buffer that stores it.

An example of a BoF induced by I/O function is shown in Fig. 3. This BoF is in the DBD::mysql module which is the perl DBI driver for mysql. In Fig. 3, the function *dbd_bind_ph* first defines an array with a constant size of 64 (line 4786). Then the function formats a

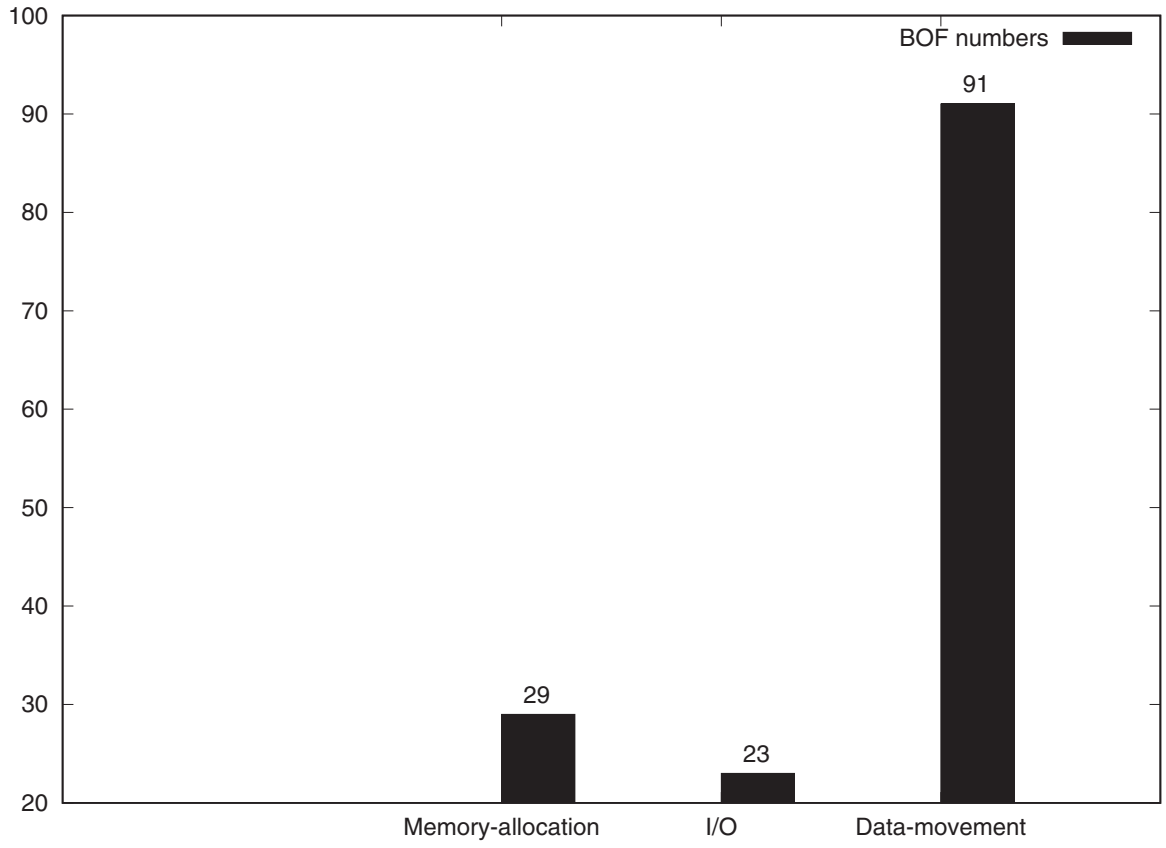


Fig. 1. BoFs induced by three types of functions.

```
202  user_payload_size = apr_request->payload_size;
203
204  aprdata = kmalloc(sizeof(struct apr_data) + user_payload_size,
205                                              GFP_KERNEL);
206
207   if (aprdata == NULL) {
208              pr_err("%s: aprdata kmalloc failed.\n", __func__);
209
210              ret = -ENOMEM;
211              goto done;
212}
213
214  voice_svc_update_hdr(apr_request, aprdata);
```

Fig. 2. A BoF induced by a memory-allocation function in *Wireshark*.

string and outputs the string to the array (lines 4830–4831). However, the string can be longer than the length of the array, and there is a BoF. This BoF would lead to a crash, causing a denial of service.

2.1.3. Data-movement

In our collection results, there are 91 BoFs induced by data-movement functions shown in Fig. 1, accounting for 63.6% of the total BoFs induced by functions, which is the most among the three types. According to statistical analysis, the data-movement functions include *memcpy*, *strcpy*, *strcat*, *strncpy*, *strncat*, etc.

After the analysis of these code segments, we get their trigger processes as follows:

1. Program moves data from source buffer to destination buffer by calling the data-movement functions, but it misses checking the size of data movement or has mistakes on the size checking.
2. Program writes data to the destination buffer, which overruns the bound of the buffer.

Fig. 4 shows a BoF induced by a data-movement function. This BoF is hidden in TigerVNC, which allows remote servers to execute arbitrary code. In Fig. 4, the program first acquires the value of *r.width()*. Then, the program performs data movement based on the calculate of *r.width()*. However, the developers miss checking the

```
4780  int dbd_bind_ph(SV *sth, imp_sth_t *imp_sth, SV *param, SV *value,
4781          IV sql_type, SV *attribs, int is_inout, IV maxlen) {
      ...                                          GFP_KERNEL);
4786   char err_msg[64];
      ...
4829   sprintf(err_msg,
4830          "Binding non-numeric field %d, value %s as a numeric!",
4831           param_num, neatsvpv(value,0));
4832   do_error(sth, JW_ERR_ILLEGAL_PARAM_NUM, err_msg, NULL);
```

Fig. 3. A BoF induced by an I/O function in *DBD::mysql*.

```
99  void ModifiablePixelBuffer::fillRect(const Rect& r, const void* pix)
    ...
105   w = r.width();
    ...
121   int  w1;
    ...
125   w1 = w;
126   while (w1--) {
127       memcpy(buf, pix, b);
128       buf += b;
129    }
```

Fig. 4. A BoF induced by a data-movement function in *TigerVNC*.

Table 3
Summary on the BoFs induced by functions.

Functions	Causes	Solutions
Memory-allocation	Allocate not enough memory.	Check the size of allocated memory.
I/O	Miss restrictions.	Check the size of the input/output content or adjust the size of buffer.
Data-movement	The moved data overruns the bounds of buffers.	Preform careful check on the bounds of buffers.

data structure of *r*, and it may be wrongful, which would cause the data movement beyond the bound of *buf*.

The BoFs induced by different functions have different causes and solutions to avoid. Table 3 summarizes the causes and solutions to the BoFs induced by the three types of functions.

2.2. Loop-BoFs

In our study, there are 68 BoFs induced by loops, accounting for 32.2% of the total BoFs. 37 of the BoFs happens because of exceeding the bounds of arrays, and others occurs due to pointer operations. The numbers of two types are very close.

After the analysis of these code segments, we get these trigger processes as follows:

1. Program performs data movement using loops from source buffer to destination buffer, but it does not check the bound of the buffers.
2. Program writes data onto the destination buffer, which overruns the bound of the buffer.

It is easy to overlook that operations on pointers and arrays, which would exceed the bounds of buffers, leading to BoFs. The loop-BoFs are elusive and easily neglected, which occur in loops that perform data movement in heap/stack memory. Thus, focusing on operation of data-movement with arrays and pointers is a good simplification for loop-BoF detection.

2.2.1. Loop-BoFs on arrays

In our study, the loop-BoFs on arrays are stack overflow. To expose the BoFs with arrays, we can analyze data-movement operations with arrays and the bound of corresponding arrays. Fig. 5 shows an example of BoF (CVE-2016-5108) due to exceeding the bound of an array. The program first defines an array with a constant bound (line 582). Then it writes data to the array in a loop (lines 592–593). However, the index (*i_ch*) of the array can exceed its bound, causing a stack overflow. More accurately, the value of $p_dec->fmt_in.audio.i_channels$ can be bigger than 2, leading to *i_ch* bigger than 1, and *channel*[*i_ch*] exceeds the bound. This BoF is very serious that can lead to denial of service or arbitrary code execution.

Table 4
Summary on the BoFs induced by loops.

Functions	Causes	Solutions
Array	Exceed the bound of an array.	Check the index of array.
Pointer	Exceed the bound of a buffer.	Perform bound checking or allocate enough size of buffer.

2.2.2. Loop-BoFs on pointers

In our study, the BoFs on pointers can be heap overflow or stack overflow, which are related to what the pointers refer to. To expose the BoFs with pointers, we can analyze data-movement operations with pointers and the allocated size of corresponding buffers. An example of a BoF with pointer is shown in Fig. 6. The program first allocates a block of buffer in heap (line 253). Then the program assigns the address of the buffer to a pointer *ptr2 257*. After that, the program performs operations on *ptr2* in a loop (308). However, the pointer *ptr2* may refer to an address beyond the buffer leading to a heap overflow. This BoF occurs in *cJSON* which would cause a denial of service (crash) or arbitrary code execution.

The BoFs induced by loops can occur on two types of objects, array or pointer. Table 4 summarizes the causes and solutions of the BoFs with the two types of objects.

3. Loop analysis

In this section, we analyze the characteristics of loops that inducing BoFs, preparing for the detection of loop-BoF. We first introduce several related definitions. Then, we present our studies on real-world loops that induce BoFs and summarize features of the real-world BoFs induced by loops.

3.1. Definitions

Definition 1 (Loop variable)**.** Given a loop $\mathcal{L}$, if the exit condition of $\mathcal{L}$ is based on the condition judgment of a variable $\mathcal{V}$, then $\mathcal{V}$ is a loop variable.

The objective to distinguish loop variable is to obtain the state of array and pointer upon loop-exit. Then we need to get iteration count of the loop. Integer variables are the most common loop variables, and these iteration numbers can be easily obtained by analyzing the integer variables. For instance, given a loop $for(int i; i < 10; i++)\{\ldots\}$, *i* is an integer variable and the exit condition is directly related to *i*, so *i* is a loop variable. *i* is also a loop variable that changes as in each iteration with the loop. In other cases, the loop variables can be non-integer variables, and to get iteration count we need to analyze the loop variables. For instance, given a loop $while((str++) != NULL)\{\ldots\}$, *str* is a string pointer and the exit condition is directly related to *str*, so *str* is a loop variable. To get iteration count of the loop, we need to calculate the initial length of the string *str*.

Definition 2 (Path)**.** Given a loop $\mathcal{L}$, if the loop block, such as the code between the braces of $\mathcal{L}$, contains a condition branch based on the condition judgment of a loop variable $\mathcal{V}$, then $\mathcal{L}$ contains a path $\mathcal{P}$.

The above definition defines the path. If a loop contains a condition branch related to the condition judgment of the loop, then the loop contains a path. In this work, the loop analysis is related to whether a loop contains paths. If a loop contains a path, we require to analyze whether the path affects the state of data movement on exit condition. For instance, in Fig. 6, *ptr* is a loop variable, and there are condition branches related to *ptr*. Therefore, this loop contains paths. In this program, the path does not affect the state

```
579 static void DecodeAdpcmImaQT( decoder_t *p_dec, int16_t *p_sample,
580                               uint8_t *p_buffer )
581 {
582 adpcm_ima_wav_channel_t channel[2];
    ...
589 for( i_ch = 0; i_ch < p_dec->fmt_in.audio.i_channels; i_ch++ )
590 {
591         /* load preambule */
592     channel[i_ch].i_predictor  = (int16_t)((( p_buffer[0] << 1 )|(  p_buffer[1] >> 7 ) ))<<7);
593     channel[i_ch].i_step_index = p_buffer[1]&0x7f;
     ...
```

Fig. 5. A BoF induced by a loop with an array in *VideoLAN VLC*.

```
234 static const char *parse_string( cJSON *item, const char *str )
    ...
253 if ( ! ( out = (char*) cJSON_malloc( len + 1 ) ) )
254     return 0;
255
256 ptr = str + 1;
257 ptr2 = out;
258 while ( *ptr != '\"' && *ptr ) {
     ...
263 switch ( *ptr )
     ...
308     default:  *ptr2++ = *ptr; break;
     ...
```

Fig. 6. A BoF induced by a loop with a pointer in *cJSON*.

Table 5
Study on real-world loops inducing BoFs.

Count	Single loop	Nested loops
No path	28	8
With path	22	10

of the data movement on exit condition. We just require to analyze the exit condition of the loop.

3.2. Analyzing loops inducing BoFs

To detect real-world BoFs induced by loops, we first analyze the loops. Code of real-world programs is very large and complex, and loops are scattered in the code. It is very difficult to precisely obtain range of every variable. So our analysis work consists of two parts: studying the characteristics of real-world loops and calculating the extreme value of pointers and array indexes based on the characteristics in the loops.

Our study on real-world loops inducing BoFs focuses on two points:

- Whether the loops are nested loops?
- Whether the loops have paths?

Table 5 presents the counts of BoFs induced by different types of loops in our study based on the two points.

In our study, there are 50 BoFs induced by single loop, accounting for 73.5% of the total BoFs induced by loops. So most of the BoFs are induced by only single loop. Other BoFs are hidden in the nested loops. In their nested loops, most of them are two-nest loops.

The counts of loops that contain paths and no path are very similar. 36 BoFs in our study have no path, 52.9% of the BoFs induced by loops. The analysis complexity of loops with paths is much more than that with no path.

The loops can be divided into four types: *SN* (single loop with no path), *SP* (single loop with paths), *NN* (nested loops with no path) and *NP* (nested loops with paths). In our work, we adopt different analysis methods for different types of loops. For SN and NN, we calculate the range of pointer or array index based on the count of iteration. For SP and NP, we adopt conservative simplified symbolic execution to get the range of pointer and array index.

Besides, real-world loops inducing BoFs have two features. First, the operations of data movement in most loops inducing BoFs are on complex data structure, such as *structs*. In our study, there are only two BoFs on the data movement with relatively simple data structure, including string array. Second, most loops inducing BoFs contain two or more than two uncertain variables, which increases the analysis complexity. In our work, we apply conservative method to analyze the cases of more than one variable.

Real-world loops are so complex that we need to consider the cases of nested loops, paths, complex data structure and multi-uncertain variables. To solve the problem with their cases, our work adopts different methods for different cases, which are presented in the following section.

4. Detection

In this section, we introduce our static loop-BoF detection technique, *DBloop*. DBloop is based on CIL. The inputs of DBloop are unmodified code.

Fig. 7 shows an overview of DBloop. DBloop consists of four steps. First, DBloop identifies the loops that contain operations of data movement with arrays or pointers which are lvalues (a lvalue refers to variable that can appear on the left side of an assignment expression and is not *const* qualified). In this step, DBloop also distinguishes loop variables for the next step. Second, DBloop applies

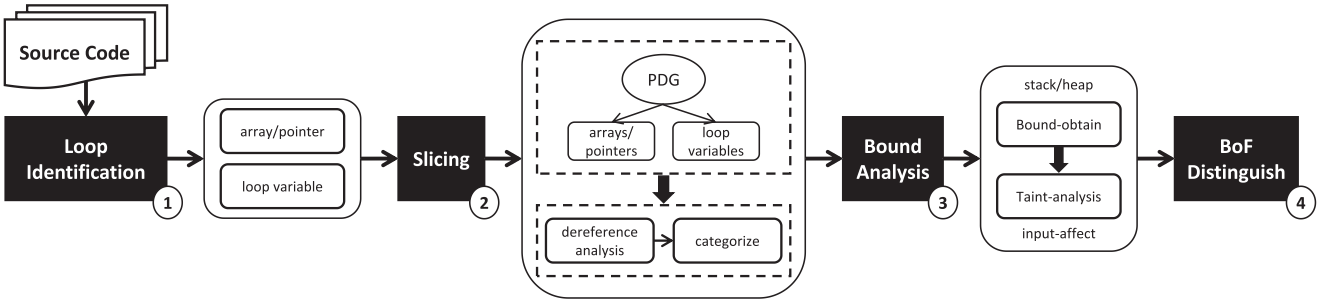


Fig. 7. Overview of DBloop.

```
loop  ::=  for mr | dowhile mr | while mr
mr    ::=  array.l | pointer.l
```

Fig. 8. Algorithm of loop identification.

code slicing on the loops, and only keeps the code related to objective arrays/pointers and loop variables. After slice, the loops will be classified into four types: SN, SP, NN and NP. Then, DBloop performs traceability analysis to obtain the bounds of buffers and taint analysis to get whether the values of array indexes or pointers can be affected by user inputs. Last, DBloop checks whether the data movement in the loops can overrun the bounds of buffers and distinguishes BoFs.

The key method of DBloop is to calculate the ranges of pointers or array indexes. For the loops with no path (SN and NN), DBloop directly calculates the maximal values of the pointers and indexes of the arrays, and check whether they can overrun the bound of the buffers. For the loops with paths (SP and NP), DBloop simplifies the target loops, translates them to SN and NN loops, and then applies the analysis as on SN and NN loops. In the following parts, we introduce each step of DBloop.

4.1. Loop identification

In general, DBloop has two important jobs in this step:

- Localizing all loops that may induce BoFs.
- Distinguishing related arrays/pointers in operations of data-movement and loop variables.

DBloop pinpoints all loops containing operations of data-movement with arrays or pointers. Fig. 8 shows the algorithm that DBloop localizes suspicious loops.

DBloop first localizes the loops, including *for*, *do-while* and *while*. One *for* loop consists of for-expressions and block-sentences. The for-expressions are presented in form of *(expressions; boolean expressions; expressions)*. The block-sentences contain operations of data movement with arrays and pointers. In the algorithm of loop identification, DBloop first identifies the loops that have the patterns, such as *for*, *do-while* and *while*. For instance, one *do-while* loop is identified as the form of *do block-sentences while (boolean-expressions)*, and one *while* loop as *while (boolean-expressions) block-sentences*. After that, DBloop will analyze the loop block to check whether there exist statements related to arrays and pointers. *array.l* and *pointer.l* represent arrays and pointers that are lvalues. There are calculation operations on indexes of arrays or pointers, and their expressions are in the block-sentences. In this way, DBloop identifies the block sentences that have operations of data movement with arrays and pointers.

The exit conditions of a loop are in Boolean expressions or in block-sentences that contain *break-sentence*. Each exit condition contains at least one Boolean expression. In this work, we say that the variables that determine the value of the Boolean expression are loop variables. The loop variables may be not so explicit. For instance, if the loop is terminated based on the content of the array, the index of the array is the loop variable. In a loop, only the indexes of arrays or pointers are related to loop variables or there are calculation operations on indexes of arrays or pointers in the block-sentences that the loop may induce BoF. DBloop will pinpoint all these loop-BoFs.

4.2. Slicing

DBloop applies slicing technique (Weiser, 1979) to simplify analysis objects. More accurately, DBloop computes all program statements that may affect the values of target arrays/pointers and loop variables. In this step, DBloop performs program slicing twice to simplify code analysis.

In the first time, DBloop slices code based on relevance and does forward slicing. In the loop identification step, DBloop localizes all target arrays/pointers and loop variables. In this step, DBloop generates a program dependence graph (PDG) (Ottenstein and Ottenstein, 1984) based on a target loop. For each slicing criterion $<q, V>$, in which q is a statement and V is a subset of variables, DBloop retains the statement only if V contains the indexes of target arrays or pointers or loop variables. After that, DBloop remains the statements related to data movement and loop iteration numbers.

In the second time, DBloop performs slicing to further simplify the analyzed code. DBloop analyzes the code of data-movement and distinguishes the dereferences of target pointers/arrays related to loop variables. After that, DBloop only remains one dereference for each array/pointer. For instance, Fig. 9 shows an example of our slicing on a real-world BoF. There is a nested for-loop in the program that uses different loop variables in different independent loops. In the loop block, there is a data-movement related to statement of array. The index of the array increases with each loop cycle. The original loop block has 15 lines of code, which includes operations of non-data-movement. DBloop then performs slicing, and only the statements related to the array and loop variable remain. More accurately, the statements of data-movement related to the array are remained. There are only 8 lines code left, which is much clearer than the original code.

In this step, DBloop needs to categorize the target loops, which is a very important job for BoF detection. After slicing, the code of loops are simpler and clearer. DBloop then classifies the loops into four types, including SN, SP, NN and NP, based on whether the loops are nested loops and whether the loops contain paths. For instance, the loops in Fig. 9 belong to NN, because the loops are nested loops that do not contain any paths.

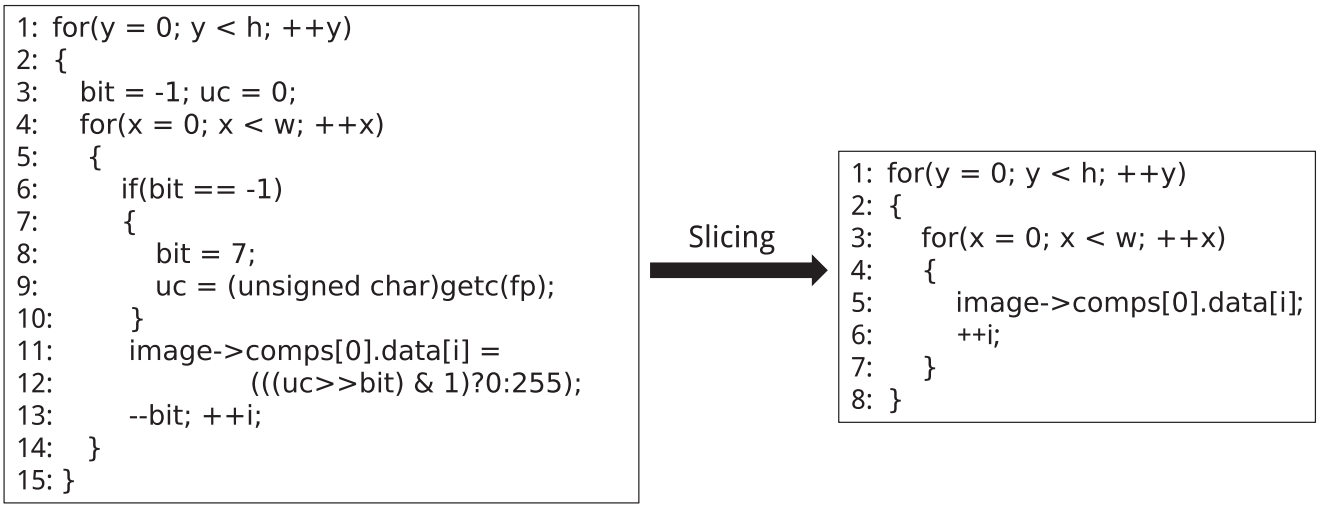


Fig. 9. Program slicing of a BoF in *OpenJPEG*.

4.3. Bound analysis

In this step, DBloop performs bottom-up analysis to achieve two objectives: getting the length of buffer and checking whether inputs can affect the numbers of iterations in loops and the lengths of target buffers. The two objectives are two basic premises of BoF detection. First, only if we obtain the length of buffer that we can judge whether one loop can induce BoFs. Then if the inputs from users can directly control the numbers of iterations in loops then the loops may induce BoFs. To achieve above two objectives, DBloop performs bound analysis, including two sub-steps, bound-obtain and taint-analysis.

In the bound-obtain sub-step, DBloop first needs to get definitions of the data structures of target arrays/pointers. DBloop finds the definitions from the current file and the included head files. Then DBloop performs bottom-up data flow analysis to get the length of buffers. In other words, DBloop finds the definitions of target arrays or the statements that the program allocates heap-memory and assigns locations to target pointers, using operations like *malloc*, *calloc*, *new*, etc. This analysis is straight-forward. For instance, after DBloop performing slicing on the loops in Fig. 9, DBloop gets that *image->comps[0].data[i]* is the target array. DBloop first finds the data definitions of *image* and *image->comps* in *openjpeg.h*, and knows that *image->comps* is an *int* pointer. Then DBloop performs bottom-up analysis to find the statements of heap-memory allocation and finds that the program uses *calloc* to allocate heap-memory and the size is *header_info.width * header_info.height * sizeof(int32_t)*.

However, it is not enough to get the size of target buffers. There are two reasons. First, the length of target buffers may be affected by more than one variable, such as the previous example, and it is hard to directly judge whether the buffer can overrun. To solve this problem, DBloop adopts BoF-distinguish analysis in the following section. Second, the length of target buffers may be affected by inputs from outside which may be uncontrolled. DBloop solves this problem in the taint-analysis sub-step.

In the taint-analysis (Ashcraft and Engler, 2002; Pratikakis et al., 2011) sub-step, DBloop performs data-flow analysis to check whether outside inputs can affect the numbers of iterations in loops and the lengths of target buffers. DBloop focuses on loop variables and variables related to the lengths of buffers. DBloop first performs bottom-up data dependence analysis to check whether there are input-statements that accept data from outside inputs and affect the values of loop variables and length of target buffers. DBloop achieves that by identifying the I/O functions, such as the read series function. If there are input-statements without value check, the relationship of data dependence is direct effect. Then the loop variables or the length of target buffers can be directly affected by outsider inputs, and DBloop will record it. For instance, the length of target buffer in Fig. 9 is determined by *header_info.width* and *header_info.height*, and after taint analysis DBloop finds that the two variables are controlled by inputs from a file. Then the length of the buffer can be affected by outsider inputs.

4.4. BoF Identification

In the last step, DBloop adopts different analysis methods for different types of loops to identify BoFs in target loops. In the slice step, loops have been divided into four types: including SN, SP, NN and NP. The loops of SN and NN do not contain any paths, and the numbers of iteration can be directly computed by exit conditions. DBloop applies algebra calculation to obtain the extremum values of indexes of target arrays or pointers, and then checks whether the buffers are overrun. The loops of SP and NP have branches in loops. DBloop first implements simplification and, then, computes the numbers of iteration to identify BoFs. For the complicated NP loops that contain multiple loop variables, DBloop applies conservative method. If DBloop cannot calculate the accurate the numbers of iteration, DBloop will report warnings. These analysis methods are presented in the following parts.

4.4.1. SN and NN

DBloop first checks whether the loop variables or length of buffer can be directly affected by outsider inputs. If the inputs can directly affect them, DBloop will report a BoF warning for the loop and record it. Otherwise, DBloop then computes the numbers of iteration. DBloop first considers the SN loop. A SN loop consists of three parts: stepping operations, an exit condition and data-movement. The stepping operations are performed on the loop variable. For a SN loop, DBloop computes stepping value on the loop variable:

$$S_f = S_f(v) \tag{1}$$

S_f is the stepping function and v is the loop variable. To calculate the number of iteration (N_i), DBloop resolves the exit condition:

$$N_i = E_f(S_f, E) \tag{2}$$

```
1: while ( *ptr != '\"' && *ptr ) {
2:             ptr++;
3:             *ptr2++ = *ptr; break;
4:             ++ptr;
5: }
```

Fig. 10. The simplified code in Fig. 6.

E is the exit condition. For a NN looop, there are more than one exit condition, loop variable and multi-level iteration. To compute number of iteration, DBloop needs to solve equations:

$$N_i = E_f(S_{f_1}, S_{f_2}, \ldots, E_1, E_2, \ldots) \tag{3}$$

After that, DBloop computes the extremum buffer access offset (value of index of target arrays or pointers) based on N_i. There are two cases for the relation between N_i and maximum buffer access offset. If the N_i directly determines the buffer access offset, DBloop can directly calculate the extremum buffer access offset based on the N_i. If the N_i indirectly determines the buffer access offset, DBloop first analyzes the relation between the N_i and the buffer access offset. It puts the N_i into the transitive relation and computes the extremum buffer access offset. DBloop performs a conservative algorithm and computes the maximum boundary for the target arrays or pointers. If one extremum value of indexes of target array or pointers is beyond the buffer, DBloop reports a BoF and records it.

4.5. SP and NP

For the loops with paths, DBloop needs to simplify target loops and remove the paths. Since the loops with paths are very complex, DBloop applies conservative method to simplify loops. The key points of the analysis are to covert the SP/NP to SN/NN.

First, DBloop computes extremum value for each loop variable based on all conditions (E_i, $i = 1, 2, \ldots$), including exit conditions, in paths. The calculation is based on boolean conditions.

$$C_i = C_f(E_i), i = 1, 2, \ldots \tag{4}$$

DBloop then computes numbers of iteration based the extremum values of loop variables.

$$N_i = E_f(S_{f_1}, S_{f_2}, \ldots, C_1, C_2, \ldots) \tag{5}$$

The following operations are similar to that of SN and NN.

Fig. 10 shows an example of simplification performed by DBloop. In the loop of Fig. 6, the branch conditions do not affect the exit conditions. DBloop compares the conditions for each branch and chooses the most conservative condition. Then the code is simplified as shown in Fig. 10. *ptr*2 is a pointer that points to a heap memory block. The length of the heap memory block is only a part of that of *ptr*. The simplified code illustrates that the length of the heap memory block can be half of that of *ptr*. Thus, DBloop will report a loop-BoF.

DBloop also considers the cases of inputs. If the inputs can directly affect the index of target array or pointer, DBloop analyzes the paths. If only the operations of data movement are in a path, DBloop reports a BoF warning. If the operations on loop variable are in paths, DBloop checks whether the extremum values overrun the buffer and the loop can be affected by inputs. If both satisfy, DBloop reports a BoF warning.

Table 6
Detecting known BOFs.

Tool	Detected BOFs	False warnings
DBloop	57	15
Splint	3	10
Checkmarx	4	26

5. Evaluation

In this section, we perform three tasks: (1) evaluating the coverage and false alarm rate of DBloop, (2) studying the loops in real-world programs, and (3) detecting unknown BOFs induced by loops in real-world programs. The first task is achieved by evaluating DBloop to detect known loop-BOFs, while compared with open-source tool, Splint (Evans and Larochelle, 2002), and commercial tool, Checkmarx. We accomplish the last two tasks by classifying loops and detecting unknown BOFs from open-source programs.

DBloop is implemented using CIL (Necula et al., 2002a) and SMT solver Z3 (de Moura and Bjørner, 2008). All experiments are conducted on a machine with 4-cores-8-threads 3.3 GHz Intel Xeon CPU (8 MB Cache) with 8 GB RAM and Linux version 2.6.32.

5.1. Detecting known loop-BOFs

We implement DBloop, Splint and Checkmarx to detect the 68 loop-BOFs collected in our study. The objective of this experiment is to evaluate the effectiveness of DBloop. We check the detected BOFs and false warnings of DBloop, while compared with Splint and Checkmarx.

Table 6 shows the detection results of the three tools. The results indicate that DBloop can detect much more BOFs than the other two tools. The detection coverage of DBloop is 83.8%, which is greater than that of Splint (4.4%) and Checkmarx (5.9%). Thus DBloop is more effective than the other tools on the loop-BOF detection. The numbers of false warnings on the detection of DBloop, Splint and Checkmrx are 15, 10 and 26, respectively. The total number of false warnings is 34, in which there are 7 false warnings that are the same between Checkmarx and DBloop. DBloop reports more false warnings only than that of Splint. The false positive rate of DBloop is 20.8%, which is reasonable on static analysis.

There are 11 BOFs that cannot be detected by DBloop. The reason is that the missed BOFs are caused by multiple factors and more analysis is required. In some of the missed BOFs, the lengths of buffers are determined by more than one variable and DBloop cannot check whether the buffers overrun. In other missed BOFs, the operations of data movement in target loops are in the form of functions, and DBloop cannot distinguish. For instance, the operations of data movement in a loop of a BOF in *Libming* are represented as *parseSWF_MORPHFILLSTYLE (f, &(fillstyle->FillStyles[i]))*, which cannot be pinpointed by DBloop.

DBloop reports 15 false warnings. DBloop would report warnings when DBloop cannot make definite judgment in the BOF detection. For instance, DBloop can detect the BOF in *ffmpeg*, but it also reports a false warning. This is because the code of *ffmpeg* contains five-layer nested loops and the index of an array is determined by four variables, and DBloop reports a BOF warning. Fortunately, these cases are not common in real-world programs. So DBloop is a very practical BOF detection tool.

5.2. Detecting unknown loop-BOFs

To detect unknown loop-BOFs from real-world programs, we implement DBloop on 20 popular open-source programs. The detection results are shown in Table 7. The 20 programs have 3,109

Table 7
Detecting unknown BOFs from real-world programs.

Programs	Total loops	SN	SP	NN	NP	Detected BOFs
util	9	3	2	0	4	0
sqlite3	380	1	13	0	9	0
lemon	84	0	9	0	4	0
deftree	31	0	6	0	3	0
base64	8	0	5	0	2	0
sws	6	0	5	0	1	1
LottoPrj	7	0	3	0	1	1
JS_Util_String	15	0	3	0	1	0
duktape	2447	0	2	0	3	0
crypt freesec	12	0	2	0	2	0
sds	12	0	2	0	2	0
slg_master	16	0	2	0	2	0
mkkeywordhash	7	0	2	0	1	0
helper	6	0	2	0	1	1
preserve	9	0	2	0	1	0
server	15	1	9	0	0	0
teamWebserver	9	0	6	0	0	1
linux_simu	7	0	5	0	0	0
lstrlib	23	0	3	0	0	0
lua	6	0	0	0	0	0
sum	3109	5	83	0	37	4

loops in total, about 155 for each. We first implement DBloop to localize suspicious loops that contain operations of data movement and classify the loops. There are 125 suspicious loops, accounting for 4.0% in total loops. The proportions of the four types of loops are 4.0%(SN), 66.4%(SP), 0%(NN) and 29.6%(NP), respectively. This classification can help us to have insight into the loops in real-world programs. The results indicate that most suspicious loops are SP-loops, and NN-loops have a low frequency in real-world programs.

The proportions of the four types of loops are SN, NN, SP and NP, respectively, in which SP-loops are the most common. However, DBloop applies an effective method to analyze SP-loops and have very high precision. Thus, DBloop is very practical in the analysis of real-world loops.

DBloop then detects loop-BOFs from the suspicious loops. We have detected 4 unknown loop-BOFs in four programs, including *sws*, *LottoPrj*, *helper* and *teamWebserver*. We have analyzed and verified each loop-BOF manually. In our experiments, the occurrence rate of loop-BOF is 0.13% in total loops. Among the 125 suspicious loops, the occurrence rate is 3.2%. Thus, the rate of loop-BOFs in the loops that perform data movement is high, which indicates that loop-BOF detection is very critical and our work is meaningful.

6. Related work

In this section, we introduce existing work on the BoF detection that can be divided into static detection techniques and dynamic detection techniques (Fang and Hafiz, 2014; Ye et al., 2016).

6.1. Static detection technique

The static analysis approaches adopt different techniques to detect BoFs from source code and binary code. The source-code based techniques (Ganapathy et al., 2003; Khedker, 2014; Larochelle and Evans, 2001; Wagner et al., 2000) scan source code to pinpoint potential code segments that are vulnerable to BoFs. They usually concentrate on data-movement library functions, such as memory copy functions, input functions and formatted printing functions. Wagner et al. (2000) propose a static technique for potential BoFs in C code that first distinguishes string operations functions, such as *strcpy*, *strcat*, *sprintf* and *gets*, and then applies static analysis to formulate the BoF detection problem as an integer constraint problem. Since the static techniques lake the run-time information, they can report many false warnings. Therefore, each reported BoF requires manual checking.

Some tools detect BoFs and reduce false warnings by matching patterns. For instance, Yamaguchi et al. (2014) propose a new representation of source code that merges abstract syntax trees, control flow graphs and program dependence graphs into a joint data structure. They apply the new data structure to represent the patterns of BoFs and then identify BoFs by matching patterns. Flawfinder (Perl et al., 2015) collects a large number of BoFs from open-source programs, and then uses these data to train patterns of BoFs. After that, Flawfinder can quickly scan source code by matching patterns to identify BoFs. Some other tools perform semantic analysis to detect BoFs. BovInspector (Gao et al., 2016) first performs warning reachability analysis on the source code. Then, it validates and classifies the reachable warning by checking the path conditions and checks the BoFs constraints. After that, BovInspector fixes the real BoFs. Splint (Evans and Larochelle, 2002) requires source annotations to identify BoFs. It models and annotates buffer sizes in the data-movement library functions, and then applies inter-procedural analysis. Marple (Le and Soffa, 2008) applies path-sensitive analysis to identify BoFs. It first identifies the infeasible paths and marks them on an intermediate representation of the program, and then identifies BoFs. It can only handle the loops with no path. However, our approach can also analyze the loops with paths. Symbolic execution is also used to detect BoFs. MergePoint (Avgerinos et al., 2014) introduces symbolic execution to analyze source code. It applies constraint solving to identify the BoFs and then verifies each possible BoF by using dynamic instrumentation.

The binary-code based detection techniques (Song et al., 2008; Wang and Lin, 2009) first distinguish data movement functions and then adopt bound analysis or symbolic execution to pinpoint BoFs. The difficulty of theses detection is the lake of data layout information. DDA (Gopan et al., 2015) adopts data-delineation analysis to distinguish top-level datum, such as an array, structure, or variable, and then applies bound analysis to pinpoint BoFs. IntScope (Wang and Lin, 2009) applies a novel method to solve data-layout problem. It first translates disassembled code into a new intermediate representation, and then performs data flow analysis and taint analysis to detect BoFs.

6.2. Dynamic detection technique

The dynamic approaches identify BoFs at run-time, which use either checking or test techniques. The checking techniques (Ding et al., 2016; Jones et al., 1997; Necula et al., 2002b) inserts checking code into programs so that BoF occurrences can be identified. More accurately, they keep check of the boundary of variables, assuming which should not be exceeded by all accesses, and report BoFs when the buffers overrun. Lhee and Chapin (2002) present a BoF detection technique that modifies the compiler to add type information and then performs bound checking at run-time. They extend the generic C compiler that augments executable files with type information. If a program has a lot of access to the buffers, such as access to arrays in loops, it will lead to heavy time overhead. HeapTherapy (Zeng et al., 2015) collects in-memory traces of target programs at runtime and detects BoFs by diagnosing the traces. HeapTherapy places inaccessible guard pages randomly throughout the heap space, so that, when the heap overruns, it is highly probable that a guard page is touched. It can be used to detect over-read attacks.

The test-based techniques apply various methods to generate inputs and then use the generated inputs to test programs and trigger BoFs. For instance, Haugh and Bishop (2003) instrument monitor code into programs to keep track of memory buffers, and design test data based on the arguments of functions to trigger

BoFs. DIODE (Sidiroglou-Douskos et al., 2015) generates inputs to trigger overflows at memory allocation sites. Some tools constraint on the input generation. For instance, Splat (Xu et al., 2008) automatically generates test inputs to trigger BoFs by applying symbolic execution to guide the test. Taintscope (Wang et al., 2010) uses a fuzzing approach to generate inputs based on dynamic taint tracing to identify effective test data. Fuzzing test has proven to be an efficient method to find BoFs in practice. TaintScope combines dynamic taint analysis and symbolic execution techniques with fuzzing test, so that, it is more efficient to generate effective inputs. SeededFuzz (Wang et al., 2016) also adopts fuzzing test to find BoFs that uses a seed selection method complementing with a seed generation method for directed fuzzing. Skyfire (Wang et al., 2017) is another fuzzing tool that applies data-driven seed generation method to generate inputs for fuzzing programs. The disadvantage of fuzzing test approaches is that they usually generates large numbers of inputs, most of which are ineffective, and require to run programs with numerous times.

6.3. Deference

The deference techniques are usually divided into three types: prevention, monitoring and input filtering. The preventation-based techniques use some methods, such as StackGuard (Cowan et al., 1998), data execution prevention (Team) and address space layout randomization (Bhatkar et al., 2003). The monitoring-based techniques monitor programs at run-time and then check them to guarantee security by applying bound checking (Akritidis et al., 2009), shadow-memory checking (Serebryany et al., 2012) or control/data flow monitoring (Zhang et al., 2013).

Inputs are the essential factors to trigger BoFs and many techniques are proposed to filter inputs (Kim, 2004; Newsome, 2006). For instance, VSEF (Newsome, 2006) filters inputs by monitoring and instrumenting the part of program execution related to BoFs.

7. Conclusion

BoF detection is of great importance for the security of software systems. Existing techniques often focus on BoFs induced by functions but ignore the BoFs by the other key factor, the loops. We first perform an empirical study on real-world buffer overflows and find out that about one third of real-world BOFs are induced by loops. We then present a novel static detection technique, DBloop, to localize loop-BOFs. According to our evaluation results, DBloop can detect more BoFs induced by loops than existing tools.

Acknowledgment

This work is supported by National 973 Fundamental Basic Research Program under grant No. 2014CB340600 and National Science Foundation of China under grant No. 61672249.

References

Akritidis, P., Costa, M., Castro, M., Hand, S., 2009. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 2009 SSYM. USENIX Association.
Ashcraft, K., Engler, D., 2002. Using programmer-written compiler extensions to catch security holes. In: Proceedings of the 2002 SP. IEEE Computer Society.
Avgerinos, T., Rebert, A., Sang, K.C., Brumley, D., 2014. Enhancing symbolic execution with veritesting. In: Proceedings of the 2014 ICSE. ACM.
base64. Available from http://flex.phys.tohoku.ac.jp/japanese/kouhou04/konnen/php-4.3.5.
Bhatkar, E., Duvarney, D.C., Sekar, R., 2003. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 2003 USENIX Security Symposium.
Checkmarx. Application Security Testing. https://www.checkmarx.com/.
Cifuentes, C., Scholz, B., 2008. Parfait: designing a scalable bug checker. In: Proceedings of the 2008 SAW. ACM.
Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., 1998. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 1998 SSYM. USENIX Association.
deftree. Available from http://ftp.mozilla.org/pub/security/nss/releases/NSS_3_2_1_RTM.
Ding, Y., Wei, T., Xue, H., Zhang, Y., Zhang, C., Han, X., 2016. Accurate and efficient exploit capture and classification. Sci. China Inf. Sci. 60, 1–17.
duktape. Available from https://github.com/civetweb/civetweb/tree/master/src/third_party/duktape-1.5.2.
Evans, D., Larochelle, D., 2002. Improving security using extensible lightweight static analysis. IEEE Softw. 19 (1), 42–51.
Fang, M., Hafiz, M., 2014. Discovering buffer overflow vulnerabilities in the wild: an empirical study. In: Proceedings of the 2014 ESEM. ACM.
crypt freesec. Available from https://github.com/facebook/hhvm.
Ganapathy, V., Jha, S., Chandler, D., Melski, D., Vitek, D., 2003. Buffer overrun detection using linear programming and static analysis. In: Proceedings of the 2003 CCS. ACM.
Gao, F., Wang, L., Li, X., 2016. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In: Proceedings of the 2016 ASE. ACM.
Gopan, D., Driscoll, E., Nguyen, D., Naydich, D., Loginov, A., Melski, D., 2015. Data-delineation in software binaries and its application to buffer-overrun discovery. In: Proceedings of the 2015 ICSE. IEEE Press.
Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H., 2013. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: Proceedings of the 2013 SEC. USENIX Association.
Haugh, E., Bishop, M., 2003. Testing C programs for buffer overflow vulnerabilities. In: Proceedings of the 2003 NDSS.
helper. Available from https://github.com/adamilab/eos.
Jones, R.W.M., Kelly, P.H.J., C, M., Errors, U., 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. Distributed Enterprise Applications. HP Labs Tech Report
JS_Util_String. Available from https://github.com/joseserverproject/JoseServer.
Khedker, U. P., 2014. Buffer Overflow Analysis for C. ArXiv e-prints.
Kim, H.-A., 2004. AutoGraph: toward automated, distributed worm signature detection. In: Proceedings of the 2004 Usenix Security Symposium.
Larochelle, D., Evans, D., 2001. Statically detecting likely buffer overflow vulnerabilities. In: Proceedings of the 2001 SSYM. USENIX Association.
Le, W., Soffa, M.L., 2008. Marple: a demand-driven path-sensitive buffer overflow detector. In: Proceedings of the 2008 FSE. ACM.
lemon. Available from https://github.com/theory/flex-lemon-example.
Lhee, K.-s., Chapin, S.J., 2002. Type-assisted dynamic buffer overflow detection. In: Proceedings of the 2002 USENIX Security Symposium. USENIX Association.
Li, L., Cifuentes, C., Keynes, N., 2010. Practical and effective symbolic analysis for buffer overflow detection. In: Proceedings of the 2010 FSE. ACM.
linux_simu. Available from https://github.com/OpenJabNab/OpenJabNab/blob/master/bootcode/compiler/mtl_linux.
LottoPrj. Available from https://github.com/FoRTe21/LottoPrj.
lstrlib. Available from https://www.lua.org/source/5.1/lstrlib.c.html.
lua. Available from https://github.com/lua/lua.
mkkeywordhash. Available from https://git.ucd.ie/ol/gridsolve.
de Moura, L., Bjørner, N., 2008. Z3: An Efficient SMT Solver. Springer, Berlin Heidelberg.
Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W., 2002a. CIL: intermediate language and tools for analysis and transformation of C programs. In: Proceedings of the 2002 CC. Springer-Verlag.
Necula, G.C., McPeak, S., Weimer, W., 2002b. CCured: type-safe retrofitting of legacy code. In: Proceedings of the 2002 POPL. ACM.
Newsome, J., 2006. Vulnerability-specific execution filtering for exploit prevention on commodity software. In: Proceedings of the 2006 NDSS.
NVD. National vulnerability database. https://nvd.nist.gov/.
Ottenstein, K.J., Ottenstein, L.M., 1984. The program dependence graph in a software development environment. In: Proceedings of the 2009 ACM SIGPLAN Notices, 19. ACM, pp. 177–184.
Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., Acar, Y., 2015. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 2009 CCS. ACM.
Pratikakis, P., Foster, J.S., Hicks, M., 2011. Locksmith: practical static race detection for C. ACM Trans. Program. Lang. Syst. 33 (1), 1–55.
preserve. Available from https://github.com/hyc/webalizer.
Rawat, S., Mounier, L., 2012. Finding buffer overflow inducing loops in binary executables. In: Proceedings of the 2012 SERE.
sds. Available from https://github.com/antirez/redis.
Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. Addresssanitizer: a fast address sanity checker. In: Proceedings of the 2012 USENIX ATC. USENIX Association.
server. Available from https://github.com/ToTenMilan/cs50-psets.
Sidiroglou-Douskos, S., Lahtinen, E., Rittenhouse, N., Piselli, P., Long, F., Kim, D., Rinard, M., 2015. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In: Proceedings of the 2015 ASPLOS. ACM, New York, NY, USA.
slg_master. Available from https://github.com/fgaud/SLG.
Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P., 2008. BitBlaze: a new approach to computer security via binary analysis. In: Proceedings of the 2008 ICISS. Springer-Verlag.
sqlite3. Available from https://www.sqlite.org.

sws. Available from https://github.com/wilbeibi/sws.
The PaX Team. Data execution prevention. https://pax.grsecurity.net/.
teamWebserver. Available from https://github.com/xiaominfc/teamtalk_websocket_server.
util. Available from https://github.com/mbmccormick/purdue.
Viega, J., Bloch, J.T., Kohno, Y., McGraw, G., 2000. ITS4: a static vulnerability scanner for C and C++ code. In: Proceedings of the 2000 ACSAC. IEEE Computer Society.
Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A., 2000. A first step towards automated detection of buffer overrun vulnerabilities. In: Proceedings of the 2000 NDSS.
Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: Data-driven seed generation for fuzzing. In: Proceedings of the 2017 SP. ACM.
Wang, T., Lin, Z., 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in $\times$86 Binary Using Symbolic Execution.
Wang, T., Wei, T., Gu, G., Zou, W., 2010. TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the 2010 SP. IEEE Computer Society.
Wang, W., Sun, H., Zeng, Q., 2016. SeededFuzz: selecting and generating seeds for directed fuzzing. In: Proceedings of the 2016 ISTASE. IEEE.
Wang, X., Chen, H., Jia, Z., Zeldovich, N., Kaashoek, M.F., 2012. Improving integer security for systems with KINT. In: Proceedings of the 2012 OSDI. USENIX Association.
Weiser, M.D., 1979. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method Ph.D. thesis.
Wikipedia. Wannacry Ransomware Attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.
Xie, Y., Chou, A., Engler, D., 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In: Proceedings of the 2003 FSE. ACM.
Xu, R.-G., Godefroid, P., Majumdar, R., 2008. Testing for buffer overflows with length abstraction. In: Proceedings of the 2008 ISSTA. ACM.
Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: Proceedings of the 2014 SP. IEEE Computer Society.
Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K., 2013. Chucky: exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 CCS. ACM.
Ye, T., Zhang, L., Wang, L., Li, X., 2016. An empirical study on detecting and fixing buffer overflow bugs. In: Proceedings of the 2016 ICST.
Zeng, Q., Zhao, M., Liu, P., 2015. HeapTherapy: an efficient end-to-end solution against heap buffer overflows. In: Proceedings of the 2015 DSN.
Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W., 2013. Practical control flow integrity and randomization for binary executables. In: Proceedings of the 2013 SP. IEEE Computer Society.

Peng Luo received the B.S. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012, where he is currently working toward the Ph.D. degree in computer science and technology. His research interests include software security and reliability in cloud computing.

Deqing Zou is a Professor of Computer Science at Huazhong University of Science and Technology (HUST), Wuhan, China. He received his Ph.D. at HUST in 2004. His main research interests include system security, trusted computing, virtualization and cloud security. He has been the leader of one "863" project of China and three NSFC (National Natural Science Foundation of China) projects, and core member of several important national projects, such as National 973 Basic Research Program of China. He has applied almost 20 patents, published two books (one is entitled "Xen virtualization Technologies" and the other is entitled "Trusted Computing Technologies and Principles") and more than 50 High-quality papers, including papers published by IEEE Transactions on Dependable and Secure Computing, IEEE Symposium on Reliable Distributed Systems and so on. He has always served as a reviewer for several prestigious Journals, such as IEEE TPDS, IEEE TOC, IEEE TDSC, IEEE TCC, and so on. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Yajuan Du received joint Ph.D. degrees of City University of Hong Kong and Huazhong University of Science and Technology in Feb. 2018 and Dec. 2017, respectively. She is now an assistant professor in School of Computer Science and Technology, Wuhan University of Technology. Her research interest focuses on optimizing access performance, data reliability and persistence of flash memories and non-volatile memories.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a Fellow of CCF, senior member of the IEEE and a member of the ACM. He has coauthored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Changming Liu received the B.S. degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2015, where he is currently working toward the M.S. degree in computer science and technology. His research interests are in the area of software security.

Jinan Shen received his M.S. degree from computer school of Wuhan University, China, in 2009. Currently he is a Ph.D. candidate at Huazhong University of Science and Technology (HUST). His research interests are in the area of security in cloud computing, focusing on privacy preserving in the cloud.