Testing Heap Overflow Bugs Caused by Integer Overflows

Chang-Hsien Tsai Shih-Kun Huang
Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{chsien,skhuang}@cs.nctu.edu.tw

ABSTRACT

In recent years, many heap overflow vulnerabilities have been found in major applications. Integer overflows are involved with these vulnerabilities and are hard to audit. We propose a new testing approach to find this kind of bug. We first analyze the program source to identify tainted data flows. We then manipulate the program to test the integrity of these suspects. With this tool, we discover two new heap overflows in Antiword.

1: INTRODUCTIONS

New vulnerabilities in software come out every day. Some of them are so notorious with which most programmers are familiar, e.g., misuse of unbounded string copy functions or format string functions. Therefore, these vulnerabilities almost disappear in major applications today [15] .

On the other hand, a new kind of vulnerability, called *integer overflow*, draws our attention. Many major applications, e.g., Microsoft Internet Explorer [22] and PHP[23], suffer from this kind of vulnerability. The vulnerability is caused by an integer overflow and then the overflowed integer is used to allocate heap memory. Because of the integer overflow, the allocated heap space is far less than what the programmer assumes, thereby causing a heap overflow later. It is hard to find these vulnerabilities manually. For example, the vulnerability presented in Section 2.1.1 had existed in the LibTIFF for more than three years before being uncovered. In this work, we propose a testing approach to find latent heap overflows in programs.

The remainder of this paper is organized as follows. In Section 2, we will cover the heap overflow and its relationship with the integer overflow. In Section 3, we detail the method of testing this vulnerability. In Section 4, we describe evaluations with case studies. In Section 5, related work will be presented. Finally, in Section 6, we present our conclusions.

2: BACKGROUND

A program allocates a section of memory as a buffer to store data. A fixed-size array is used if a programmer knows how much memory he needs. However, some memory requirements are unknown until runtime, and the program must allocate sufficient memory on the fly. Many library calls come in handy for this purpose, such as the `malloc()` in the C library. The dynamically allocated memory resides in a memory area, called *heap*.

Although the buffer is allocated dynamically with the appropriate size, the space is also finite. So, the heap is also vulnerable to buffer overflow. If a program writes data beyond the boundary of the buffer, a heap overflow occurs.

The heap overflow is also exploitable---execution of arbitrary code, although one requires more skills to exploit it than the stack overflow. However, stack overflow vulnerabilities are so notorious that most of them have been fixed in major software packages. On the other hand, heap overflow vulnerabilities emerge recently. These two kinds of buffer overflows are exploitable because they store user data and control data nearby. Overflows in the user data can overwrite the control data. Therefore, data from malicious users can take over the control of the program via the buffer overflow.

In most implementations of heap management, each chunk of memory is connected as a linked list [1]. There are metadata in the head of each chunk of memory. Overflows in a chunk would overwrite the metadata of the next chunk. Once upon freeing the next chunk, the system will maintain the heap structure according to the metadata. If the metadata is overwritten by malicious data, control data in the memory, such as return addresses and the *GOT table*, can be overridden via the metadata.

2.1: INTEGER OVERFLOWS

We survey many heap overflow vulnerabilities and identify the most common cause. For flexibility, programs normally allocate as much memory as users want. For example, a program reads an image file and dynamically allocates sufficient memory according to the header of the image. If the length and width specified

in the header are less than needed, the allocated memory is insufficient. This would introduce a heap overflow when the program reads the image into the allocated memory.

Integer overflow can also contribute to a heap overflow if the integer is used with dynamic memory allocation. The integer type can represent a finite space of numeric value. The space is determined by the size and signedness of the integer type. For example, a 32-bit unsigned integer can only represent from `0x00000000` to `0xffffffff`. When an integer increments over its maximum possible value, an integer overflow occurs. The 32-bit integer, `0xffffffff`, plus one becomes `0x00000000`, i.e., zero.

At first glance, integer overflow seems not be a serious bug. The overflowed integer may just produce weird results and thus easily manifest itself. Nevertheless, if the integer represents the size of a buffer in the dynamic allocation, this may introduce a vulnerability. The overflowed integer becomes zero or a small number, which is far less than the desired size of the buffer. The heap overflow occurs when the buffer is used as its originally assumed size.

Most programmers intuitively assume that requesting zero size would cause the memory allocation to fail and return errors, so they do not check for this case. In fact, if the requested size is zero, the behavior is implementation-defined. Either a null pointer is returned, or the behavior is as if the size were some nonzero value. Unfortunately, some implementations choose the latter one.

For example GNU C library [12], a request for zero bytes (i.e., `malloc(0)`) returns a pointer to a block of memory whose size is 16 bytes. Most programmers are not aware of this and check the return value for null pointer. In next sections, we present three examples of heap overflow vulnerabilities, which are caused by the integer overflow.

2.1.1: CASE STUDY (I). There is a heap overflow vulnerability in LibTIFF, which supports for the Tag Image File Format (TIFF). The overflow occurs in the `TIFFFetchStripThing()` function. The number of strips (`nstrips`) is used directly in the `CheckMalloc()` function without checking. The program calls

```
malloc(user_supplied_int*size(int32))
```

with `0x40000000` as the user supplied integer. There is an integer overflow, and `malloc()` is called with a length argument of `0`. This has the effect of returning the smallest possible `malloc()` chunk. A user controlled buffer is subsequently copied to this small heap buffer, causing a heap overflow. This bug can be exploited to overwrite heap structures and to execute any injected code.

2.1.2: CASE STUDY (II). There is an integer overflow in the GD graphics library leading to a heap overflow [21]. The vulnerable code locates in the function `gdImageCreateFromPngCtx()` of the file `gd_png.c`. The function loads a PNG file into GD data structures. The problem occurs when allocating memory for the image rows. Two user supplied values are multiplied together (`rowbytes * height`), and used to allocate memory for an array of pointers. If a user loads a malicious PNG image, the system may be under control by the image's creator.

2.1.3: CASE STUDY (III). The remote RTSP heap overflow in MPlayer [24] is another typical example. MPlayer dynamically allocates a buffer:

```
  description =
malloc( sizeof(char)*(size+1));
```

The `size` comes from the statement:

```
size = atoi( rtsp_search_answers(
             rtsp_session,
             "Content-length"));
```

, which reads data from remote users and converts it into an integer. So this is also vulnerable to heap overflow caused by integer overflow.

2.2: INPUT VALIDATION

This kind of heap overflow can be avoided by using accurate *input validation*. Almost all programs receive input from the outside. Browsers read web pages from web servers via the socket. Editing tools read documents from the file system. Console programs read options from the command line. All these programs accept the input data and perform their work. However, if they use the input data to perform important operations, they must check it first. The check process is called input validation. Programmers must assume all the input data are malicious. For the integer type of input, the maximum and minimum acceptable values must be checked with. All values outside of the legal range are rejected.

However, input validation is usually absent, because there are so many integers that programmers often forget to check some of them. Sometime, input validation is done but not correctly (e.g., not checking maximum value, or not checking minimum value, or both). Thus, we propose a testing mechanism to check all suspect integers for heap overflows.

3: METHOD

Our approach is based on both static analysis and dynamic analysis. First, we use static analysis to find all

tainted paths in the program. These paths are suspects for heap overflow. Then we test the tainted path with different integers and observe the results.

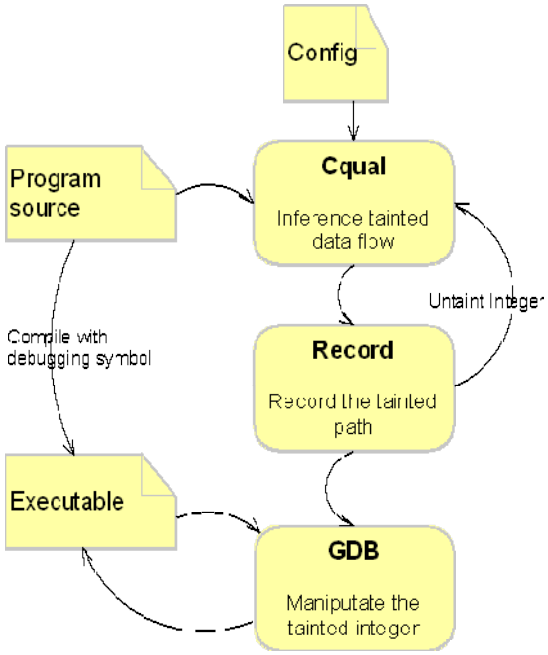

Figure 1. the architecture of our analysis tool

3.1: TAINTED PATHS

Programs read the data through the I/O functions, such as `fgets()` and `fread()`. The input is stored in the buffer and waits for processing. Programs would read some integers from the buffer and compute an appropriate size for memory allocation. The paths from I/O functions to memory allocation functions are called *tainted paths*.

All `malloc`-related functions that are concerned with the input are suspects for heap overflow. The sources of tainted paths are I/O functions and the sinks of tainted paths are `malloc`-related functions.

As Figure 1 shows, we use CQual [10] to determine all tainted paths. CQual is a type-based analysis tool for specifying and checking properties of C programs. CQual extends the type system of C with extra user-defined type qualifiers. We use two new types: *untainted* and *tainted*. In the configuration of CQual, we mark all data that input functions read as *tainted*, and `malloc`-related functions should accept *untainted* data.

The configuration files define the function prototypes annotated with the new types. For example, `fread()` is declared as

```
int fread(void $tainted *ptr, int size,
          int nitems, FILE *stream);
```

The input data read from file are declared as tainted. And `xmalloc()` is declared as

```
void $_1 * xmalloc($untainted size_t i);
```

The parameter of `xmalloc()` must be untainted data. Figure 2 is a tainted path that violates the above rules. The source of the dataflow is `fread()`, which reads data into the buffer `aucBytes` in line 193 of the file `misc.c` and the sink is `xmalloc()` in line 570 of the file `summary.c`.

Of all the variables in the tainted paths, the first integer in the path is critical, because we can manipulate it to test the possible heap overflow. We can find the integer after the cast operation. In this example, the first integer, `tLen` in `summary.c:558`, is called the *critical integer*. We will manipulate the critical integer later in the GDB.

```
prelude.cq:41  $tainted <= *fread_arg1
    misc.c:193          <= *aucBytes
  summary.c:545         <= *aucBuffer
  summary.c:558         <= cast
  summary.c:558         <= cast
  summary.c:558         <= tLen
  summary.c:570         <= tLen@570
  summary.c:570         <= xmalloc_arg1
prelude.cq:105          <= $untainted
```

Figure 2. a tainted path found in the Antiword

CQual reports one tainted path each time. We must untaint the tainted integer to get another tainted path. The process is repeated until CQual does not report any tainted path. In this way, we can collect all tainted paths in the test application.

3.2: INTEGER MANIPULATION

With all these tainted paths, we then test whether they are vulnerable. On each tainted path, input data are passed through many variables. Based on our surveys about heap overflows, we observe that the critical integer is the source of the heap overflow. Not all integers in the program need to be tested, but the critical integer needs to be tested.

We use GNU debugger, GDB [11], to execute the program with different test cases. Breakpoints are set on the next line after the assignment of the critical integer. When GDB stops at the breakpoint, it will set the critical integer with a new value with the command:

```
set var = value
```

Therefore, in each iteration of test, we manipulate the integer with all possible variables to simulate the possible malicious data. Then we observe how much

memory `malloc()` allocates and the result of the test application.

There are three possible results of the tainted path.

1. The program crashes, and allocated memory in `malloc()` is insufficient. The program may be exploitable.
2. The result of the program is incorrect. There may be a bug in the tainted path.
3. The result of the program is correct. Then, we need to set another value to the critical integer at the breakpoint and observe the result of the test application. If all possible values of the critical integer work fine in the test application, we can assume that the tainted path is not vulnerable to heap overflow.

3.3: TESTING COVERAGE

The accuracy of testing is limited by the testing coverage. The testing coverage of our work is determined by the test cases. Assume we have *M* suspicious `malloc`-related functions and the *i*th test case triggers C_i among the tainted paths. Then we have the testing coverage:

$$\frac{\bigcup C_i}{M}$$

Because of the popularity of testing-driven development, most software has its own testing cases, which have very high testing coverage. Therefore, we have no or little effort to create test cases used in our experiments.

4: EVALUATIONS

To evaluate the functionality of our tool, we test with Antiword [26] and find two new heap overflow bugs. Antiword is a free MS Word reader that converts the binary files from Word to other formats, e.g., plain text, PostScript, etc. We test the Antiword 0.37, which is the latest version.

First, we use CQual to analyze Antiword. CQual totally reports 99 tainted paths in Antiword, and we can easily trigger 16 of them with a simple word file. For each tainted path, GDB manipulates the critical integer in the tainted path, observes the argument of the `malloc()`, and watches the result of Antiword.

We find that two tainted integers would cause Antiword crash out of the sixteen suspects. These two tainted paths shows in Figure 3 and 4. In the tainted path of Figure 3, data is read by the `fread()` in the line 193 of the file `misc.c`. The tainted data is read into an integer in the line 615 of file `stylesheet.c`. GDB sets the critical integer `tStshInfoLen` as zero. Finally, the tainted integer becomes the argument of the `xmalloc()` in the line 636 and the Antiword crashes in the end.

With the tainted path in Figure 4, GDB executes Antiword and sets the breakpoint in line 289 of the file `prop8.c`. The critical integer `tBytes` is set as zero. The critical integer becomes the argument of the `xmalloc`() without changing its value. In the end Antiword crashes.

```
 prelude.cq:41    $tainted <= *fread_arg1
     misc.c:193            <= *aucBytes
   wordwin.c:182           <= aucHeader[]
   wordwin.c:200           <= *aucHeader
properties.c:119           <= *aucHeader
stylesheet.c:615           <= cast
stylesheet.c:615           <= cast
stylesheet.c:615           <= tStshInfoLen
stylesheet.c:636           <= xmalloc_arg1
  prelude.cq:105           <= $untainted
```

Figure 3. a tainted path of Antiword in stylesheet information handling.

```
prelude.cq:41  $tainted <= *fread_arg1
    misc.c:193          <= *aucBytes
    misc.c:240          <= *aucBuffer
   prop8.c:285          <= aucTmp[]
   prop8.c:288          <= cast
   prop8.c:288          <= cast
   prop8.c:288          <= cast
   prop8.c:288          <= __cst@288
   prop8.c:288          <= tBytes
   prop8.c:291          <= xmalloc_arg1
prelude.cq:105          <= $untainted
```

Figure 4. a tainted path of Antiword in section property information handling

We observe the file offset of the critical integer in the Word file with the help of `ftell()`. The crafted word files that cause the critical integer overflow can also trigger the above two crashes, so they are real vulnerabilities.

5: RELATED WORK

A considerable amount of work has been performed on detecting program errors and identifying their root causes either by static analysis, or by observing their running behavior through dynamic program instrumentation. In this section, we review different work in each category.

5.1: STATIC ANALYSIS

Static analysis [5, 9, 28] bases on the information provided by the source code without actually executing the program. It may check the vulnerable call sequence or function. The main limitation is that with static analysis it is hard to infer the information known at runtime. Our method combines the static analysis and dynamic analysis to find out real vulnerabilities.

5.2: MEMORY CHECKER

Memory checker is another useful debugging tool. *CRED* [20] is an extension of *GNU C compiler* to track the referent object of each pointer. *Purify* [13] is a famous commercial software to detect heap overflow, among other memory errors. It can discover various bugs or suspicious things, while a program is running, such as reading or writing to memory where it should not read or write, using uninitialized variables. *Valgrind* [17] emulates the x86 CPU and runs the program binary directly. *Memcheck* plug-in in Valgrind takes the similar approach to Purify and enhances its detection to bit-level.

5.3: ABNORMALITY DETECTOR

Buffer overflow exploits must transfer the control of program by overwriting important variables. Observing these abnormal changes, one can detect buffer overflow exploit and terminate the vulnerable program. Several works design compiler extension to add this checking. *StackGuard* [7] adds *canary* between return address and saved base pointer. *SSP* [8], originally named *propolice*, and Microsoft Visual Studio /GS option [4] adds canary between the saved base pointer and local variables. Before any user function returns, its canary is checked to detect buffer overflow. *StackShield* [27] uses *Ret Range Check* to protect the return address. It saves return the return address of the current function in another global variable. When the function returns, it matches the return address with the stored return address. *Binary Rewriting* [18] protects the integrity of the return address on the stack by modifying the binary code. It adds the same detection mechanism similar to StackGuard without source code.

Software wrapper is another approach to monitor dangerous library call. *Libsafe* [2] wraps dangerous functions (such as `strcpy()`, `strcat()` and etc.) to enforce boundary checking. Wrapping functions compute the size between the buffer starting address and saved frame pointer. If the input data is larger than the size, libsafe halts the program to avoid overwriting the saved base pointer and the return address. Robertson et al. [19] wraps heap-related functions to detects heap overflow. By wrapping `malloc()`, it inserts canary and padding in front of each memory chunk. By wrapping `free()`, it checksums the chunk to ensure the canary unchanged. *STOBO* [14] wraps user functions to detect buffer overflow.

5.4: AVOIDANCE OF EXPLOITS

Exploits usually execute the injected malicious code, thus without executing these code the system cannot be compromised. Linn et al. [16] observe that successful exploits must invoke system calls. They instrument a white list, which records the program counter with system call, in the executable. The kernel can use the list to differentiate user code from injected code at runtime.

Address space layout randomization (ASLR) [25] is another technique to avoiding exploits. By moving the code segment, stack segment and other segments, to different address, ASLR can hinder attackers to predict the exploit address. PointGuard [6] encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. Hence, the injected value by attacks will become useless. RISE [3] *XOR* trusted binary code with random number during loading and XOR back in instruction fetch time. Malicious code injected without XORing becomes garbage and soon crashes.

Non-executable stack [25] makes the user stack become readable and writable only, thus injected code in the stack cannot execute. However, the integrity of the return address is not checked, so this technique can be bypassed by jumping to the existing code in the heap or code segment, being known as *return to libc* attack.

Although these techniques are good for security, they do not solve the bug itself. Most work make control inception become *Denial of Service*.

6: COUNCLUSION

We have proposed a testing approach to find out heap overflows caused by integer overflows. The vulnerability arises from allocating less memory space, because the parameter passed to the `malloc` function is inappropriate. The shortfall is usually the result of an integer overflow.

In our testing approach, all possible tainted paths are discovered from the program source. We then test each possible tainted path by manipulating the critical integer. We minimize the number of test by feeding with possible value. Our tool finds two new bugs in Antiword.

This vulnerability can be avoided by performing input validation. However, programmers usually forget to filter the input integer. Even they do filter the input integer, the filter criteria is often imperfect. With our tool, these latent bugs can be found and the software quality is enhanced.

REFERENCES

[1] Anonymous. Once upon a free(). *Phrack Magazine*, 10(57), 2001.

[2] A. Baratloo, T. Tsai, and N. Singh. Libsafe: Protecting critical elements of stacks. White paper, Bell Labs, Lucent Technologies, Dec. 1999.

[3] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication*

security, pages 281–289. ACM Press, 2003.
[4] B. Bray. Compiler security checks in depth. Technical report, Microsoft Corporation, 2002.
[5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In V. Atlury, editor, *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS-02)*, pages 235–244, New York, Nov. 18–22 2002. ACM Press.
[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104. USENIX, Aug. 2003.
[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stack-Guard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.
[8] H. Etoh. Gcc extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/.
[9] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan./Feb. 2002.
[10] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.
[11] GNU. Gdb: The gnu project debugger. http://www.gnu.org/software/gdb/.
[12] GNU. Gnu c library. http://www.gnu.org/software/libc/libc.html.
[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136, 1992.
[14] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the 2003 Symposium on Networked and Distributed System Security*, Feb. 2003.
[15] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. "noir" Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*. John Wiley Sons, 2004.
[16] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, and J. H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 2005 USENIX Security Symposium*, pages 239–254, July 2005.
[17] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
[18] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, June 2003.
[19] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *proceedings of 17th USENIX Large Installation Systems Administration (LISA) Conference*, Oct. 2003.
[20] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, Feb. 2004.
[21] SecuriTeam. Gd graphics library integer overflow leading to a heap overflow. http://www.securiteam.com/unixfocus/6L00N2ABFK.html, Oct. 2004.
[22] SecurityTracker. Microsoft internet explorer integer overflow in processing bitmap files lets remote users execute arbitrary code. http://securitytracker.com/id?1009067, Feb. 2004.
[23] SecurityTracker. Php emalloc() integer overflow may let remote users execute arbitrary code. http://securitytracker.com/id?1009653, Apr. 2004.
[24] Mplayer - the movie player. http://www.mplayerhq.hu, 2000.
[25] Documentation for the pax project. http://pax.grsecurity.net/docs/index.html.
[26] Antiword: a free ms word document reader. http://www.winfield.demon.nl/.
[27] Vendicator. Stack shield:a "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/, Jan. 2000.
[28] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. Tokenbased scanning of source code for security problems. *ACM Transactions on Information and System Security*, 5(3):238–261, Aug. 2002.