2022 6th International Conference on System Reliability and Safety

Runtime Recovery for Integer Overflows

Zhen Huang
School of Computing
DePaul University
Chicago, USA
zhen.huang@depaul.edu

***Abstract*—Despite decades of effort in research and engineering, integer overflows remain a severe threat to software security. Many tools are developed to detect integer overflows at runtime. However, the vast majority of them terminates program execution when an integer overflow is detected. This essentially causes denial-of-service, which is undesirable in many scenarios in practice. We propose a recovery mechanism designed for safe recovery from integer overflows. The recovery mechanism detects integer overflows and rectifies the values involved in arithmetic operations causing integer overflows so that it prevents the occurrence of the integer overflows and enables the program to continue execute safely. We have designed and developed a tool called RIO that can automatically synthesize and instrument our recovery mechanism into target programs. Our evaluation shows that RIO can successfully synthesize and instrument the recovery mechanism into programs containing real world vulnerabilities and the instrumented recovery mechanism allows the programs to recover safely in the face of exploits intending to trigger the vulnerabilities.**

***Index Terms*—System Recovery, Fault Recovery, Software Reliability, Software Vulnerability, Integer Overflow, Program Analysis, Static Taint Analysis.**

I. Introduction

Integer overflows is one of the most common vulnerabilities [1] and one of the top 25 dangerous software weakness [2]. They are often exploited by attackers to trigger software faults such as buffer overflows and null-pointer dereference to compromise computer systems or cause denial-of-service.

As it is crucial to address integer overflows, many tools have been proposed to detect and fix integer overflows [3]–[14]. Because statically identifying integer overflows tend to produce high false positives, a plethora of integer overflow detection tools focus on runtime detection. They usually detect integer overflows using standard tests on integer overflow conditions that indicate whether an integer overflow occurred. On the contrary, the main challenge of fixing integer overflows lies in what to do when an integer overflow is detected. Most tools simply terminate program execution or rely on developers to specify a generic operation that should be executed when any integer overflow occurs [3], [11]–[13].

One approach to address the challenge is to divert program execution to error handling code when integer overflows occur. Prior work has shown that existing error handling code in a program can be automatically identified [15]. Its limitation is that only 75.1% of the functions of target programs contain error handling code that can be automatically identified. The rest of the functions either do not have any error handling code or cannot be identified easily.

To address these limitations, we propose a recovery mechanism that can enable programs to safely recover from integer overflows. In the face of an input that is about to trigger an integer overflow in an arithmetic operation, our recovery mechanism rectifies the operands of the arithmetic operation so that it not only prevents the occurrence of the integer overflow but also enables the program to continue execute safely, without losing program availability.

We have designed and implemented a software tool called RIO. It uses static taint analysis to automatically identifies arithmetic operations for which program input can cause integer overflows, synthesizes the recovery mechanism, and instruments the recovery mechanism into target programs.

Unlike prior work that recover from software faults via an external program execution monitor that nullifies operations leading to software faults and thus incurs high runtime overhead, RIO instruments target programs with lightweight recovery mechanism to enable the programs to recover from integer overflows.

In summary, we make the following contributions in this paper:

- We present a novel recovery mechanism that enables existing programs to recover safely from inputs intending to trigger integer overflows.
- We describe the design and implementation of RIO, our software tool for automatically synthesizing and instrumenting the recovery mechanism into target programs.
- We evaluates the effectiveness of RIO and the safety of the recovery mechanism. RIO successfully synthesizes and instruments the recovery mechanism for the arithmetic operations involved in 8 of 10 real world vulnerabilities. The recovery mechanism allows programs to safely recover from exploits intending to trigger these 8 vulnerabilities.

The organization of the rest of this paper is as follows. Section II discusses related work. Section III describes the attack model. We illustrate an example usage of RIO in Section IV. In Section V we describe our design and implementation. Following that, we present our evaluation results in Section VII and discussions in Section VI. We conclude in Section VIII.

978-1-6654-7091-9/22/$31.00 ©2022 IEEE

II. RELATED WORK

A. Recovering from Faults

Over the decades many different approaches have been developed to provide fault recovery. A common approach is to create checkpoints during program execution and rolls back to a checkpoint when a fault is triggered, in order to restore a system to the last-known good state [16]–[18]. Its drawback is that both the creation of checkpoints and the rollback typically incur high runtime overhead.

Another approach to maintains system functionality after the occurrence of a fault is to undo the side-effects of the operations triggering the fault [19]. However, it is challenging to identify or design appropriate undo operations [20].

Other approaches discard fault-triggering operations and manipulate program state in a way to allow a program to continue executing safely [21]–[25].

These approaches generally require the use of an external program execution monitor to sandbox the program execution after they change program state for fault recovery. As an example, recovery shepherding uses an external program execution monitor to recover from divide-by-zero and null-dereference errors [23]. When a fault is detected, recovery shepherding starts the program execution monitor to change the program state and track the data affected by the change. Until all the affected data are discarded by the program, it uses error containment to prevent any of the affected data from being written to the file system.

B. Detecting and Fixing Integer Overflows

A large body of work has been proposed to detect, mitigate, or fix vulnerabilities, particularly integer overflows [3]–[14], [26]–[29].

IntPatch automatically fixes Integer Overflow to Buffer Overflow vulnerabilities in C/C++ programs at compile time [11]. It utilizes type theory and static dataflow analysis to identify potential vulnerabilities and then instruments runtime checks on integer overflows.

IntScope performs symbolic execution and taint analysis on binaries to detect the program paths that can propagate integer overflows into exploitable vulnerabilities [6]. It symbolically executes x86 binary code and explores every branch in a target program when the branch is feasible. The symbolic execution identifies integer overflows whose results can reach sensitive points, including memory allocation, memory access, branch statement, and program-specific points.

Using dynamic analysis, IntTracker detects Integer-Overflow-to-Buffer-Overflow (IO2BO) vulnerabilities in C/C++ programs [12]. It aims to reduce the false positives in detecting integer overflows by delaying the detection to the locations where the result of an integer overflow will be used for memory operations. By doing so, it will not report the occurrence of an integer overflow if the result of the integer overflow is never used for memory operations.

SIFT uses symbolic execution to learn symbolic conditions on inputs to detect integer overflows [10]. Starting from critical sites in a target program, including memory allocation and block copy sites, SIFT uses backward static analysis to generate symbolic conditions that are conjunctions of functions on input fields. A symbolic condition for a critical site denotes how the value used at the critical site is derived from input fields. Based on symbolic conditions, SIFT generate filters to filter out inputs that will cause overflowed values being used at critical sites.

SOAP rectifies inputs that could trigger integer overflows into benign inputs [30]. It learns the constraints for benign inputs by running target programs using benign test inputs. If an input violates any of these constraints, SOAP rectifies the content of the input so that the input satisfies the constraints.

Talos [15] and RVM [27] prevent vulnerabilities from being exploited by instrumenting Security Workaround for Rapid Response (SWRR) into target programs. Each SWRR safely disables the execution of a vulnerable function by diverting program execution to error handling code when a vulnerable function is called, so that the vulnerability cannot be triggered.

III. ATTACK MODEL

In our attack model, an attacker tries to exploit an integer overflow vulnerability in a program in order to hijack program execution, gain unauthorized access to the computer running the program, or cause the program to be offline, i.e. denial-of-service attack. We assume that the attacker does not have physical access to the computer, and does not have the privilege to terminate the execution of the program. The attacker can access the program only by sending inputs to the program.

A. Hijacking Program Execution

The attacker can craft a malicious input to trigger an integer overflow in the program and consequently causes a software fault that can be exploited to divert program execution. For example, an integer overflow can cause the program to allocate a smaller-than-needed buffer and subsequently lead to a buffer overflow that can be used to overwrite function return address to a value supplied by the attacker for code injection attack [31] or return-oriented-programming attack [32].

B. Unauthorized Access

An integer overflow in the program can be exploited by the attacker to gain unauthorized access to sensitive data on the computer, also called *information disclosure*. As an example, a calculated value may be used as an index to access data, but an integer overflow occurred in the calculation can make the index points to protected data that is not supposed to be accessed by the attacker [33].

C. Denial-of-Service

The input supplied by the attacker causes an integer overflow in the program execution and either causes the program to terminate abnormally, e.g. fail an assertion, or triggers a software fault, which cannot be used by the attacker to hijack program execution or gain unauthorized access. As a result, the program is offline and the service provided by the program is no longer available.

IV. RIO

We design RIO to enable programs to recover from integer overflows gracefully. In this section we illustrate how RIO can be used for gocr, an optical character recognition (OCR) program, to recover from a real-world integer overflow.

A. An Example Integer Overflow

As shown in Figure 1, function readpgm in gocr is in charge of reading the content of an image file. The function contains an integer multiplication at line 15 involving variable nx and ny, which are the width and height of the image, read from the image file at line 7 and 8 respectively. The multiplication can cause an integer overflow.

```
1  void readpgm(char *name, pix* p, ...){
2     FILE *f1;
3     int nx, ny, i, k;
4     unsigned char *pic;
5
6     f1 = fopen(name, ”rb”);
7     nx = read_int(f1);
8     ny = read_int(f1);
9+    int size = nx * ny;
10+   if (nx != 0 && size / nx != ny) {
11+      log_message(”Integer␣overflow␣occurred!”);
12+      nx = ny = 0;
13+      RIO_start_recovery();
14+   }
15    pic=malloc(nx * ny);
16    fread(pic, nx, ny, f1);
17    fclose(f1);
18    p->p=pic;
19    p->x=nx;
20    p->y=ny;
21 }
22
23 int otsu(unsigned char *image, int dx, int dy) {
24    unsigned char *np;
25    int i, j, k;
26    int ihist[256];
27
28    ....
29    for (i =  0; i <  dy; i+=k) {
30      np = &image[i * cols];
31      for (j = 0; j < dx; j++) {
32        ihist[*np]++;
33        ....
34      }
35    }
36    ....
37 }
```

Fig. 1. An integer overflow vulnerability in gocr, adapted from CVE-2005-1141. Lines prefixed by ”+” are instrumented by RIO for recovery from the integer overflow.

The result of the multiplication is used as the size to allocate a memory buffer by calling malloc. Function readpgm then reads all the lines of the image into the buffer at line 16. At last, it passes the width and height of the image, back to its caller via the pointer parameter p at lines 18–20.

Function otsu calculates an image intensity threshold by analyzing the image data stored in the buffer allocated and filled by function readpgm. Its parameters dx and dy are the image width and height returned from function readpgm.

To exploit the integer overflow, an attacker can craft an image file that containing a huge number, such as 1,073,741,825, as the image height and number 4 as the image width. This will cause an integer overflow at line 15 and thus allocate a smaller-than-needed buffer. When function otsu tries to analyze the image data in the buffer, the dereference of pointer np at line 32 will trigger a memory fault. We note that an attacker may also craft an image file that triggers the same integer overflow but causes a memory fault at line 16.

B. Recovery Mechanism

A user can enable a target program like gocr to defend against such an attack by using RIO to instrument our recovery mechanism into the code of the target program. The recovery mechanism consists of two components, *data rectification*, which nullifies the effect of an integer overflow, and *error containment*, which prevents the data rectification from causing persistent data corruption.

For the example integer overflow, RIO will instruments lines 9–14 into function readpgm. Lines 9–12 correspond to data rectification, while line 13 starts the error containment.

C. Data Rectification

Line 9 performs the same multiplication as line 15 so that line 10 can check whether an integer overflow occurred during the multiplication. If an integer overflow occurred, line 12 rectifies variables involved in the multiplication, nx and ny, to zeroes and line 13 calls an API function in RIO's runtime library to enable error containment.

The rectification of variables involved in the multiplication will has the effect of bypassing the pointer dereference at line 32 in function otsu, because it nullifies the image width and height, which are passed to variable dx and dy that dictate the execution of line 32. As a result, no software fault will happen in the face of the integer overflow.

RIO disables the error containment when a recovery finishes. It considers a recovery finishes when the program execution reaches an annotated program location.

D. Error Containment

An acute reader may have the concern that it is possible for the data rectification to cause unexpected program behaviors, particularly corrupting persistent data. To address this concern, RIO uses error containment to prevent any data affected by the rectification from being written to the file system.

RIO intercepts all system calls that could send data out of the running process of a target program. Similar to [23], it disallows writes to files. A user can also configure RIO to log the data intended for the skipped file writes during the error containment. The logged data can then be examined by an user and committed to the file system with user approval.

V. DESIGN AND IMPLEMENTATION

A. Overview

RIO protects a target program from integer overflows by generating and instrumenting our integer overflow recovery mechanism into the target program. The recovery mechanism includes data rectification code to rectify values involved in integer overflows and error containment code to ensure that the data rectification will not lead to persistent data corruption.

As shown in Figure 2, RIO takes a target program as input, generates and instruments the recovery mechanism into the target program in three phases.

The first phase analyzes the code of target program and outputs a list of arithmetic operation locations and the list of data dependent on the values involved in each arithmetic operation, called influenced data. It consists of one step: *identifying arithmetic operations*.

The second phase takes the output from the first phase as input and outputs synthesized data rectification code for each arithmetic operation identified in the first phase. This phase has one step: *synthesizing data rectification code*.

The third phase takes the data rectification code synthesized in the second phase as input. It instruments both the data rectification code and the error containment code into the target program. It includes two steps: *instrumenting data rectification code* and *instrumenting error containment code*.

We have implemented a prototype of RIO for C/C++ programs. Our implementation is based on LLVM [34], a popular compiler infrastructure.

B. Identifying Arithmetic Operations

RIO identifies seven types of arithmetic operations in a target program. These arithmetic operations and their corresponding overflow conditions are listed in Table I.

We focus on arithmetic operations whose operands are derived from user inputs, because user inputs may trigger integer overflows in them. RIO uses static taint analysis [9] to identify these arithmetic operations.

Initially the static taint analysis associates a taint with user input data. It then propagates the taint to instructions and other data dependent on user input data by following the data dependency in the target program recursively.

RIO adds each tainted arithmetic operation into a list of identified arithmetic operations. The operands for these arithmetic operations are deemed as derived from user input.

For the code in Figure 1, the static taint analysis associates an initial taint with `f1`, the file descriptor for a user input file. Because function `read_int` returns an integer read from `f1`, the taint is propagated to `nx` and `ny`, and then to the multiplication at line 15 as `nx` and `ny` are the operands for the multiplication. Consequently this multiplication is added to the list of identified arithmetic operations.

The output of this step is the list of the identified arithmetic operations. The second phase will synthesize data rectification code for each arithmetic operation in the list.

C. Synthesizing Data Rectification Code

This step takes the list of identified arithmetic operations as input, and outputs synthesized data rectification code for each arithmetic operation in the list.

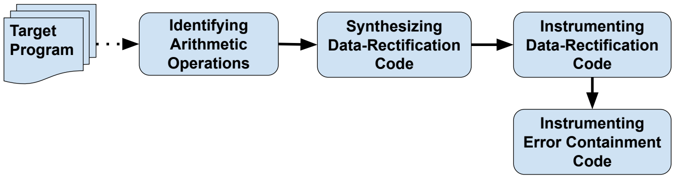
Target Program
Identifying Arithmetic Operations
Synthesizing Data-Rectification Code
Instrumenting Data-Rectification Code
Instrumenting Error Containment Code

Fig. 2. Workflow of RIO: each rounded rectangle represents a step in RIO; dotted lines denote input data; solid lines denote the order of steps.

TABLE I
OVERFLOW CONDITION FOR INTEGER OPERATIONS.

Type	Arithmetic Operation	Overflow Condition
Signed	$c = a + b$	$(a > 0 \land b > 0 \land c < a) \lor$ $(a < 0 \land b < 0 \land c > a)$
Unsigned	$c = a + b$	$c < a$
Signed	$c = a - b$	$(a < 0 \land b > 0 \land c > 0) \lor$ $(a > 0 \land b < 0 \land c < 0)$
Unsigned	$c = a - b$	$a < b$
Signed	$c = a * b$	$a \neq 0 \land c/a \neq b$
Unsigned	$c = a * b$	$a \neq 0 \land c/a \neq b$
	$c = a \ll b$	$c \gg b \neq a$

The data rectification code is in charge of testing whether an integer overflow occurred and rectifying relevant values if that is the case. It also logs a message indicating that data rectification has taken place so that the user is aware of that. Lastly it enables the error containment after rectifying values.

To determine whether an integer overflow occurred for the corresponding arithmetic operations, the data rectification code checks the overflow condition listed in Table I, based on the type of the arithmetic operation.

RIO clones the arithmetic operation and checks the cloned arithmetic operation before the identified arithmetic operation, rather than doing the check after the identified arithmetic operation. This is because we would like to make use of the identified arithmetic operation after rectifying the involved values. For example, RIO makes a clone of the multiplication at line 15 and places it at line 9, in Figure 1.

If check uses an overflow condition requiring the use of the result of an arithmetic operation, RIO also creates a new variable to hold the result of the cloned arithmetic operation, such as the new variable `size` at line 9.

One challenge for our design is to decide what values should the used for data rectification. For integer overflows in real world programs, we find that the result of the arithmetic operation and the involved values are often used as either a loop upper bound or a length argument used by functions manipulating character strings or similar types of data.

Illustrated in Figure 3, the subtraction at line 9 can cause an integer overflow and the result of the subtraction is passed as the length argument to copy a string. While in the code in Figure 1, the values involved in the multiplication at line 15 are used as loop upper bounds at line 29 and 31 respectively.

This observation makes us consider using zero as the result of an arithmetic operation in which an integer overflow will occur. In other words, the data rectification code should rectify

```
1   handle_t http_response_prepare(connection *con) {
2       ....
3       size_t plen = (size_t)(qstr - pstr);
4       size_t rlen = strlen(con->request.uri);
5+      if (plen > rlen) {
6+          log_message("Integer␣Overflow␣occurred!");
7+          rlen = 1; plen = 0;
8+      }
9       strncpy(con->query, qstr + 1, rlen - plen - 1);
10      ....
11  }
```

Fig. 3. The result of an arithmetic operation is used in a string manipulation function, adapted from CVE-2019-11072 in lighttpd, a popular web server. Lines prefixed by '+' are instrumented by RIO for integer overflow recovery.

the values involved in an arithmetic operation in a way so that the original arithmetic operation will produce a result of zero.

RIO uses a Satisfiability Modulo Theory (SMT) solver to find out how to rectify operands involved in an arithmetic operation. It creates a formula based on the arithmetic operation and lets the SMT solver produce operand values that can satisfy the expected result value.

As an example, for the subtraction at line 9 in Figure 3, RIO passes the formula $\texttt{rlen} - \texttt{plen} - 1 == 0 \wedge \texttt{rlen} >= 0 \wedge \texttt{plen} >= 0$ to the SMT solver Z3 and receives a solution $(\texttt{rlen} == 1, \texttt{plen} == 0)$.

One potential issue with this choice is that a division by zero fault can be triggered if the value rectified to zero happens to be used as a divisor after the rectification. To avoid this issue, RIO statically checks whether the result or operands of the arithmetic operation is used as a divisor. If so, RIO will not synthesize data rectification code for the arithmetic operation.

RIO *cannot* synthesize data rectification code in two cases. First, it cannot synthesize data rectification code when the arithmetic operation involves function calls. If data rectification code only changes the return value of the function call without changing the value of the arguments passed to the function call, the program may be left in an inconsistent state.

Second, it cannot synthesize data rectification code if the values to be rectified cannot be propagated to affect its dependent data. Rectifying the values without rectifying their dependency can leave the program in an inconsistent state.

This step produces the data rectification code that RIO successfully synthesizes for each identified arithmetic operation.

D. Instrumenting Data Rectification Code

Taking the synthesized data rectification code and the locations of the corresponding arithmetic operations, this step instruments the data rectification code into a target program.

It iterates through each arithmetic operation, and instruments the data rectification code corresponding to the arithmetic operation right before the program location of the arithmetic operation.

E. Instrumenting Error Containment Code

The error containment code consists of a runtime library and the code to initialize the runtime library and to enable or disable the error containment. This step instruments both of them into a target program.

The runtime library is implemented as a shared library for intercepting target program's output to the file system via the write system call. When error containment is enabled, it examines the filename associated with the file descriptor passed to any call to write to determine whether the call intends to output to a real file in the file system. If that is the case, the call will be ignored so that no output will be made into the file system during error containment.

The runtime library also provides a set of API functions for initializing the runtime library, and enabling or disabling error containment. To initialize the runtime library when the program starts running, RIO instruments a call to the runtime library's initialization API function into the entry point of the target program, such as the main function.

The code to disable the error containment is instrumented at an annotated program location in the target program, typically the location where the target program receives a new user input, such as a new network request or new user input file. As a result, the error containment is disabled when a new input is received by the target program.

VI. Discussions

RIO rectifies values involved in the arithmetic operations causing integer overflows to make the arithmetic operation produces a zero. Our evaluation shows that the rectification is safe. While it is possible that the rectification might cause errors in certain target programs, prior work [21], [23] have also shown that it is safe to rectify fault-related values to zeros.

Our prototype of RIO is designed to provide safe recovery for every arithmetic operation whose operands come from user inputs. This strategy works on the LLVM bitcode compiled from the source code and offers full-fledged protection, but the protected programs will incur the runtime overhead for checking integer overflows for these operations.

It should be noted that our recovery mechanism can be instrumented directly into the binary code of a deployed program as a temporary fine-grained patch against exploits to integer overflow vulnerabilities, before vulnerability patches are applied. We plan to explore this patch-based strategy by adopting source-code-to-binary-code matching [35].

VII. Evaluation

In this section, we evaluate the safety of the recovery mechanism generated and instrumented by RIO. We first present the overall evaluation results and then use case studies to discuss the details for each evaluated case.

All our evaluations are conducted on a workstation equipped with an 3.60GHz Intel Core i7-7700 CPU and 16GB RAM. The workstation runs 64-bit Ubuntu 18.04 desktop operating system on a 2TB 7200 RPM SATA hard drive.

A. Recovery Safety

We use real world integer overflow vulnerabilities in popular Linux programs for our evaluation. The majority of the vulnerabilities are evaluated in our prior work [9]. In total, we evaluate 10 integer overflow vulnerabilities from 8 different programs including an IRC server, a web server, a programming language interpreter, an OCR program, and four image processing libraries or tools. Table II lists the name, type, and the number of source code lines for each program. In total the programs have 880,677 lines of source code.

TABLE II
BENCHMARK PROGRAMS.

Program	Type	SLOC
ngircd	IRC server	14,660
lighttpd	web server	61,234
php	programming language interpreter	556,363
gocr	OCR tool	30,654
fig2dev	image processing tool	41,834
libftiff	image processing library	57,415
exiv2	image processing library	84,490
libsixel	image processing library	34,027

Table III lists the integer overflow vulnerabilities. The arithmetic operations involved in the vulnerabilities consist of 2 additions, 3 subtractions, and 5 multiplications. For each vulnerability, we first evaluate whether RIO can synthesize and instrument the recovery mechanism into the program in which the vulnerability exists. Then we run a proof-of-concept exploit to trigger each integer overflow and verify whether the instrumented recovery mechanism can allow the program to safely recover from the integer overflow.

RIO successfully synthesizes data rectification code for 80% of the vulnerabilities. For 2 of the vulnerabilities, RIO is unable to synthesize data rectification code.

With the recovery mechanism generated and instrumented by RIO, the programs containing these vulnerabilities no longer raise software faults when processing the exploits intending to trigger the vulnerabilities.

B. Case Studies

CVE-2005-0199. `ngircd` is an IRC server. The integer overflow vulnerability involves a subtraction: $\texttt{sizeof(TheMask)} - \texttt{strlen(at)} - 4$. RIO is unable to synthesize data rectification code for the subtraction because it involves a function call to `strlen`.

TABLE III
RECOVERY FOR INTEGER OVERFLOW VULNERABILITIES. COLUMN "ARITH." DENOTES THE TYPE OF EACH ARITHMETIC OPERATION.

CVE#	Program	Arith.	Rectified?	Safety?
2005-0199	ngircd	-	N	N/A
2005-1141	gocr	*	Y	Y
2006-2025	libftiff	+	Y	Y
2006-4812	php	*	N	N/A
2007-1001	php	*	Y	Y
2019-11072	lighttpd	-	Y	Y
2019-13109	exiv2	-	Y	Y
2019-13110	exiv2	+	Y	Y
2019-19746	fig2dev	*	Y	Y
2019-20205	libsixel	*	Y	Y

CVE-2005-1141. This is an integer overflow vulnerability in `gocr`, an OCR tool. The integer overflow occurs in a multiplication: $\texttt{nx} * \texttt{ny}$. RIO generates data rectification code to set both `nx` and `ny` to zeroes.

CVE-2006-2025. This integer overflow vulnerability is in `libtiff`, an popular image processing library. The involved arithmetic operation is an addition: $\texttt{dir->tdir\_offset} + \texttt{cc}$. The data rectification synthesized by RIO sets both `dir->tdir_offset` and `cc` to zeroes.

CVE-2006-4812. This is an integer overflow vulnerability in `php`, the official PHP programming language interpreter. The integer overflow involves a multiplication: $\texttt{size} * \texttt{nmemb}$. RIO cannot synthesize data rectification code for it, because the two variables, `size` and `nmemb`, are function parameters passed by value. As a result, the rectification on them cannot be propagated to any value dependent on the arguments corresponding to them.

CVE-2007-1001. This integer overflow vulnerability is also in `php`. It involves a multiplication: $\texttt{wbmp->width} * \texttt{wbmp->height}$. RIO synthesizes and instruments data rectification code to set both variables to zeroes.

CVE-2019-11072. `lighttpd`, a widely-used web server, contains the vulnerability. A subtraction, $\texttt{rlen} - \texttt{plen} - 1$, can cause the integer overflow. The data rectification code synthesized by RIO sets `rlen` to 1 and `plen` to zero so that the result of the subtraction is zero.

CVE-2019-13109. `exiv2` is an image processing library containing the vulnerability, which involves a subtraction: $\texttt{chunkLength} - \texttt{iccOffset}$. For this vulnerability, RIO synthesize data rectification code that sets both `chunkLength` and `iccOffset` to zero.

CVE-2019-13110. This is another vulnerability in `exiv2`. It involves an addition: $\texttt{o} + 2$. The data rectification code synthesized by RIO sets `o` to zero.

CVE-2019-19746. This vulnerability is in an image processing tool called `fig2dev`. The vulnerability involves a multiplication: $2 * \texttt{type}$. RIO synthesizes data rectification code to set `type` to zero.

CVE-2019-20205. `libsixel`, an image processing library, has this vulnerability, which involves a multiplication: $\texttt{width} * \texttt{height} * 3$. RIO synthesizes data rectification code to set both `width` and `height` to zero.

VIII. CONCLUSION

In this paper we propose a novel recovery mechanism for arithmetic operations that can cause integer overflows. The recovery mechanism rectifies operands of arithmetic operations that can cause integer overflows in order to nullify the effect of the integer overflows. We have implemented a tool called RIO to automatically synthesize and instrument our recovery mechanism into target programs. We evaluate RIO on real world integer overflow vulnerabilities and it successfully synthesizes the recovery mechanism for 80% of them. The recovery mechanism synthesized by RIO enables the programs to safely recover from exploits that trigger these integer overflows.

ACKNOWLEDGEMENT

This work was supported in part by US National Science Foundation (NSF) grant CNS-2153474.

References

[1] "CWE Definitions," https://www.cvedetails.com/cwe-definitions.php, 2020.

[2] "Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2020.

[3] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert, "Intrepair: Informed repairing of integer overflows," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2225–2241, 2021.

[4] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 760–770.

[5] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and C. M. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," *ASPLOS*, pp. 473–486, 2015.

[6] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution." in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*. The Internet Society, 2009. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2009.html#WangWLZ09

[7] Y. Zhang, X. Sun, Y. Deng, L. Cheng, S. Zeng, Y. Fu, and D. Feng, "Improving Accuracy of Static Integer Overflow Detection in Binary," in *Research in Attacks, Intrusions, and Defenses*, H. Bos, F. Monrose, and G. Blanc, Eds. Cham: Springer International Publishing, 2015, pp. 247–269.

[8] R. E. Rodrigues, V. H. Sperle Campos, and F. M. Quintao Pereira, "A fast and low-overhead technique to secure programs against integer overflows," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*, 2013.

[9] Z. Huang and X. Yu, "Integer overflow detection with delayed runtime test," in *Proceedings of the 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, ser. ARES 2021. ACM, 2021, pp. 28:1–28:6. [Online]. Available: https://doi.org/10.1145/3465481.3465771

[10] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound Input Filter Generation for Integer Overflow Errors," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 439–452. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535888

[11] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *Computer Security – ESORICS 2010*, D. Gritzalis, B. Preneel, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 71–86.

[12] H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 483–494. [Online]. Available: https://doi.org/10.1145/2714576.2714605

[13] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "IntPTI: Automatic integer error repair with proper-type inference," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 996–1001.

[14] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 792–801.

[15] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 618–635.

[16] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.

[17] Z. Huang and D. Lie, "Ocasta: Clustering configuration settings for error recovery," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014, pp. 479–490.

[18] N. S. Bowen and D. K. Pradham, "Processor-and memory-based checkpoint and rollback recovery," *Computer*, vol. 26, no. 2, pp. 22–31, 1993.

[19] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo workarounds for kernel bugs," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2381–2398. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/talebi

[20] A. B. Brown and D. A. Patterson, "Undo for operators: Building an undoable e-mail store," in *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. San Antonio, TX: USENIX Association, Jun. 2003. [Online]. Available: https://www.usenix.org/conference/2003-usenix-annual-technical-conference/undo-operators-building-undoable-e-mail-store

[21] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., "Enhancing server availability and security through failure-oblivious computing," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 21–21.

[22] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: Surviving and preventing memory management bugs during production runs," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 159–172. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519083

[23] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594337

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 87–102. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629585

[25] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 782–791. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486891

[26] A. Aumpansub and Z. Huang, "Detecting software vulnerabilities using neural networks," in *Proceedings of the 13th International Conference on Machine Learning and Computing, Shenzhen China, 26 February, 2021- 1 March, 2021*, ser. ICMLC 2021. ACM, 2021, pp. 166–171. [Online]. Available: https://doi.org/10.1145/3457682.3457707

[27] Z. Huang and G. Tan, "Rapid Vulnerability Mitigation with Security Workarounds," in *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research*, ser. BAR '19, February 2019.

[28] Z. Huang and M. White, "Semantic-aware vulnerability detection," in *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2022, pp. 68–75.

[29] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. Frans Kaashoek, "Improving integer security for systems with KINT," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, 2012, pp. 163–177.

[30] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 80–90.

[31] Aleph One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[32] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[33] C. Wellons, "Integer overflow into information disclosure," https://nullprogram.com/blog/2017/07/19/, 2021.

[34] "The LLVM Compiler Infrastructure," http://llvm.org/, 2022.

[35] D. Miyani, Z. Huang, and D. Lie, "BinPro: A Tool for Binary Source Code Provenance," *arXiv*, 2017.