# Security-oriented Program Transformations to Cure Integer Overflow Vulnerabilities

Zack Coker
Auburn University
zfc0001@tigermail.auburn.edu

**Categories and Subject Descriptors:** D.1.2 [Automatic Programming]: Program transformation; D.2.9 [Software Management]: Software maintenance

**General Terms:** Security.

**Keywords:** Program Transformation, Integer Overflow.

## 1. Introduction

When integer operations in C mix the type of operands, the result may be unexpected. There are three possile vulnerabilities [2]: 1) a *signedness bug* occurs when an unsigned type is interpreted as signed, or vice versa; 2) an *arithmetic overflow* occurs when integer operations such as addition or multiplication produce a result that overflows the allocated storage; 3) a *widthness bug* is the loss of information when a larger integer type is assigned to a smaller type, e.g., int to short.

We describe three program transformations that cure the three types of integer overflow vulnerabilities. We implemented these program transformations on top of CR-12: our Eclipse-based infrastructure to develop refactorings for C programs. Our program transformations successfully removed all integer overflow vulnerabilities from 7,147 benchmark programs of NIST's SAMATE reference dataset [5]. They also remove real vulnerabilities from real programs. For space issues, we describe here the experience of applying them on one software—libpng.

## 2. Three Program Transformations

### 2.1 Add Integer Cast (AIC)

**Context:** Add Integer Cast transformation removes signedness bugs by explicitly adding type casts. Consider the program from the SAMATE reference database [5].

```
short data;
...
data = -1;
char dest[100] = "";
if (data < 100) {
   ...
   memcpy(dest, src, data);
```

A negative number for the variable `data` wrongly bypasses the conditional test. In the memcpy function, the value of `data` will be converted to a very large unsigned integer value. This will cause the program to write past the bounds of the char array `dest`.

**Mechanism:** A user selects an integer variable and invokes the AIC transformation. The program transformation determines the type of a variable (signed short in the example), checks if it is used in an unsafe context (signed short compared with an unsigned integer in the comparison expression), and explicitly add type casts.

```
short data;
...
data = -1;
char dest[100] = "";
if ((unsigned int) data < 100) {
   ...
   memcpy(dest, src, data);
```

### 2.2 Remove Artithmetic Operator (RAO)

**Context:** One of the most common vulnerabilities in C code are arithmetic overflows. Consider the vulnerability in rdesktop 1.5.0 [6]. In iso.c file, function iso_recv_msg contains:

```
s = tcp_recv(s, length - 4)
```

If length is less than 4, the subtraction silently wraps around producing an invalid/unexpected value.

**Mechanism:** A user selects an arithmetic operation, involving +, -, *, /, ++, –, +=, -=, *=, or /= operators, and invokes the transformation. The RAO transformation determines if the integer operation is unsafe and replaces it with safe functions from CERT's IntegerLib library [8]. The safe functions introduce callbacks that are explicitly invoked when an overflow occurs.

```
s = tcp_recv(s, ui2us(subui(length , 4)))
```

### 2.3 Integer Type Change (ITC)

**Context:** ITC corrects a variable's declared type. Consider this recent vulnerability in libpng v1.4.9 [7].

```
int copy = output_size - count;
if (avail < copy) copy = avail;
png_memcpy(output + count, png_ptr->zbuf, copy);
```

`copy` is declared a signed integer but `png_memcpy` requires an unsigned integer as its third argument. A negative value of `copy` will cause a buffer overflow in `png_memcpy`.

**Mechanism:** A user selects an integer variable and invokes ITC transformation. The program transformation determines the type of the variable, checks if it is used as another type in important contexts (e.g., when a variable is an lvalue or when a variable is used in a function call, etc.), and modifies the type to the appropriate type.

```
unsigned int copy = output_size - count;
if (avail < copy) copy = avail;
png_memcpy(output + count, png_ptr->zbuf, copy);
```

## 3. Evaluation

We implemented the transformations as Eclipse plugins. We did not use CDT, since CDT lacks sophisticated analyses. The transformations were developed on CR-12, an in-house framework for building program transformations for C programs. CR-12 handles C preprocessor issues that complicate refactoring implementations, and provides sophisticated analyses such as name binding analysis, type analysis, control flow analysis, and most importantly data flow analysis.

The transformations were validated in two ways. To validate that they successfully cure vulnerabilities, we applied them on NIST's SAMATE reference dataset. To validate that they do not break existing code, we applied them on real software—we describe the experience from libpng here.

### 3.1 SAMATE

SAMATE [5] is a joint project of NIST and DoHS. It has a suite of test bench programs in C, C++, and Java to demonstrate common security problems. For C\C++ code, the test suite version 1.0 provides 45,324 test programs for 116 different Common Weakness Enumerations (CWE). The transformations were applied on the relevant CWE-s. AIC was tested on CWE 194 (1296 programs) and CWE 195 (1296 programs), RAO on CWE 190 (2430 programs), CWE 191 (810 progams), and CWE 680 (324 programs), and ITC on CWE 196 (19 programs) and CWE 197 (972 programs). Our transformations successfully removed all vulnerabilities.

### 3.2 libpng

libpng, a graphics library, had a recent vulnerability in version 1.4.9 [7] that was fixed by ITC (section 2.3).

We ran the transformations repeatedly on libpng v1.4.9. The transformations were applied to preprocessed programs: in total 16 programs with 50,758 LOC.

AIC was applied on all local variables, all formal parameters, all array access expressions and all structure element access expressions—1830 in total. It changed 490 expressions that used unsafe local variables, 240 expressions that used an unsafe parameter, 83 expressions with an unsafe array access, and 547 expressions with an unsafe structure element access. In total, 1360 explicit cast were added.

RAO applied to all binary expressions, prefix expressions, postfix expressions, and arithmetic assignment expressions (e.g., +=)—11,849 in total. It changed 1,452 expressions: 1,310 binary expressions, 2 prefix expressions, 62 postfix expressions, and 78 arithmetic assignment expressions.

ITC was applied on all local variables. It checked 474 local variable declarations and changed 135.

To check that the program transformations do not break the code, we compiled the three resultant programs, one each from all invocations of AIC, RAO and ITC. The resultant programs passed libpng's test suite. While this is not a formal proof, correct test results even after a large volume of changes suggest that the transformations are robust.

## 4. Related Works

Most of the existing works on integer overflow vulnerability concentrate on detecting the vulnerability. Many sophisticated program analysis tools [1] [3] [9] are available for automatically detecting these vulnerabilities in the source code. There are compiler based detection tools, such as gcc with `-ftrapv` option, which forces gcc compiler to insert additional calls (e.g., `_addvsi3`) before signed addition operations to catch overflow errors.

Our approach provides power tools for developers to remove integer vulnerabilities. They are similar to refactorings [4], but they do not intend to preserve behavior. They improve the security of systems, which means they preserve expected behavior, but remove the unintended vulnerability.

## References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[2] blexim. Basic integer overflows. *Phrack*, 60, 2002.

[3] X. L. David Molnar and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[4] M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.

[5] National Institute of Standards and Technology (NIST). SAMATE - Software Assurance Metrics and Tool Evaluation. `http://samate.nist.gov/Main_Page.html`, 2012.

[6] National Vulnerability Database. CVE-2008-1801, 2008.

[7] National Vulnerability Database. CVE-2011-3026, 2012.

[8] R. Seacord. *CERT C Secure Coding Standard*. Addison-Wesley, 2008.

[9] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.