An 'Explicit Type Enforcement' Program Transformation Tool for Preventing Integer Vulnerabilities

Munawar Hafiz

Department of Computer Science and Software Engineering
Auburn University
Email: munawar@auburn.edu

Abstract

A security-oriented program transformation is similar to a refactoring, but it is not intended to preserve behavior. Instead, it improves the security of systems, which means it preserves the expected behavior, but changes a system's response to security attacks. This demo is about a tool for *Explicit Type Enforcement* transformation, which adds proper typecast to integer variables. The tool is built using Eclipse CDT and applies on C programs. Preliminary results show that it is very effective in fixing integer-related vulnerabilities. Power tools such as these can improve developer productivity and produce vulnerability-free software.

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. Integer Overflow Vulnerability

When arithmetic operations mix the type of operands, the result may be unexpected. For example, a signedness bug occurs when an unsigned variable is interpreted as signed, or vice versa [2]. This type of behavior can happen because a computer makes no distinction between the way signed and unsigned variables are stored. Hence, the end result is not the one originally intended. If an unsigned integer is compared to a signed integer in a program, an attacker can carefully inject inputs to bypass the comparison. Similarly, if signed values are used in an arithmetic operation, an attacker may cause an overflow and store a value with a wrong sign. When this is used in an unsigned context, an error will occur.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

Consider this very simple program snippet.

```
1  ...
2  char buf[80];
3  int s;
4  s = atoi(argv[1]);
5  if (s >= 80)
6     return -1;
7  memcpy(buf, argv[2], s);
8  ...
```

This very simple and apparently innocuous C program has a serious integer overflow vulnerability. The variable s is a signed integer, but it is compared with an unsigned integer in line 5. An attacker can bypass that check by providing a negative input, e.g., -1. Then in line 7, the signed negative value of s will be converted to a large unsigned value. Hence an attacker can load a large malicious payload in the buffer in line 7. A carefully crafted payload can cause a buffer overflow attack, and can allow the attacker to take control of the system.

Such integer related vulnerabilities have become very prominent in recent years.

2. Motivation for a tool

Most of the existing works on integer overflow vulnerability concentrate on detecting the vulnerability. Many sophisticated program analysis tools [3] [1] [4] 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. Also there are integer overflow detection tools that apply on binaries [7, 8].

While determining whether a particular vulnerability exists in source code is very important, we believe that researchers stop one step short of solving the problem, because they still require a programmer to manually add security checks in the program to prevent the attack.

Our research on program transformations explores how existing security solutions could be automatically introduced to an application to retrofit 'security on demand' [5]. We have been describing the mechanism of program transformations so that tools could be built for software developers [6],

similar to refactoring tools. This demo describes a tool for preventing integer related vulnerabilities.

3. *Explicit Type Enforcement* Program Transformation Tool

An *Explicit Type Enforcement* program transformation explicitly casts the type of operands so that they are properly handled in an operation.

In order to apply the program transformation, a developer has to specify a target input variable.

The program transformation tool checks whether the input variable has been used in an unsafe context. It explicitly casts the type of variables in an arithmetic operation.

A developer can apply an *Explicit Type Enforcement* transformation on the variable s in the listing in section 1. It explicitly forces typecasting in all the places the variable s is used.

```
1   ...
2   char buf[80];
3   int s;
4   s = (signed) atoi(argv[1]);
5   if (((unsigned) s) >= 80)
6      return -1;
7   memcpy(buf, argv[2], (unsigned) s);
8   ...
```

In line 5, the comparison mixes a signed integer with an unsigned integer. By explicitly promoting the type in the comparison operation, the program transformation ensures that an attacker cannot bypass the operation by passing a signed value.

4. Content of the Demo

During the demo, the presenter will introduce the tool and the research behind it. Then he will run the tool on open source C programs of varying size. He will run test cases on the unmodified and modified programs and demonstrate the similarity of their 'good path behavior'.

Some of the sample programs will have known integer overflow vulnerabilities. In that case, the presenter will run exploit codes on unmodified programs, apply program transformation on variables, and run exploit codes on modified programs. The exploit code will be ineffective on the the modified program.

A video demo of the tool is available at the author's webpage: `https://netfiles.uiuc.edu/mhafiz/www/research/sopt/AddIntCast.avi`.

5. Under the Hood

The program transformation is written in Java using Eclipse CDT framework. CDT has a program analysis package called CODAN that provides limited control flow analysis. The tool uses this to search for all instances of a chosen integer variable. The scope is always within a procedure. When all the instances of an integer variable is found, the tool analyzes the context of each instance to identify whether the integers have been used properly. The tool keeps a ranking of all possible integer variables. The ranking is used to determine whether the integer usage has been proper. Whenever, integers of different rankings are used, the proper ranking is introduced as a type cast.

6. Conclusion

Security-oriented program transformations are real because they combine human ingenuity with the thoroughness of computers. The analysis and transformation performed by the tool is done in many places of a program. It is tedious and error-prone for a developer to manually check so many instances. Instead, developers remain at the policy level and the tool implements the structural change.

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*, page 143, Washington, DC, USA, 2002. IEEE Computer Society.

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

[3] E. N. Ceesay, J. Zhou, M. Gertz, K. N. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. In R. Büschkes and P. Laskov, editors, *DIMVA*, volume 4064 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2006.

[4] X. C. 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*. USENIX, Aug. 2009.

[5] M. Hafiz. *Security On Demand*. PhD thesis, University of Illinois Urbana-Champaign, 2010.

[6] M. Hafiz, P. Adamczyk, and R. Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS-09)*, Feb 2009.

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

[8] R. Wojtczuk. UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries. In *22nd Chaos Communication Congress*, 2005.

Presenter Biography. Munawar Hafiz is an Assistant Professor at Auburn University, AL. His research is on leveraging program analysis and program transformation techniques to improve application security. He has prototyped several program transformation tools applicable to C and Java [6], including this tool for preventing integer overflow attacks. Munawar has presented his research in various forms at previous SPLASHes, e.g., as part of a tutorial, as a poster, and as a finalist project in ACM student research competition.