Finding Vulnerabilities in Software Ported from 32 to 64-bit CPUs [*]

Ibéria Medeiros, Miguel Correia
Universidade de Lisboa, Faculdade de Ciências, LASIGE
ibemed@gmail.com, mpc@di.fc.ul.pt

Introduction Manufacturers like Intel and AMD started commercializing processors with 64-bit architectures a few years ago. Most code running in these processors today was initially written for 32-bit processors. However, when porting software written in C language from one architecture to another that represents numbers with a different number of bits it is easy to introduce bugs and even security vulnerabilities. This abstract shows how these vulnerabilities can be introduced, presents a source code analysis tool that finds such vulnerabilities and summarizes the results of applying the tool to test more than 20 open source applications with a total of more than 4.5 million lines of code.

The main problem when porting C programs from 32 to 64-bits CPUs is the following. Programs in C for 32-bit CPUs follow the ILP32 model [4]. In this model, three types have a size of 32 bits: *int*, *long* and *pointer*. This fact is well-known by programmers that abuse it to rather freely exchange data between variables of these three types. When programs with such liberties are ported for 64 bits and no care is taken to deal with the lack of conversion, vulnerabilities can be introduced. For 64-bit CPUs three models are used. In the LP64 model the *int* type remains with 32 bits but *long* and *pointer* are upgraded to 64 bits. This is the model adopted by Unix variants, including Linux, and open source software. In the LLP64 model, *int* and *long* remain with 32 bits and the *pointer* type changes to 64 bits. This model was adopted by Microsoft. In the ILP64 model the three types are upgraded to 64 bits. This model is used in some mainframes and supercomputers (e.g., Cray).

The problem appears mostly in LP64 because *int* and *long* have different sizes. Therefore, an assignment of a value of type *long* to an *int* variable or a *pointer* to an *int* causes a *truncation*, which is a well-known category of *integer overflow* vulnerability [1]. An example of vulnerability of this category is the SSH CRC-32 compensation attack detector vulnerability that appeared in 2001 (CVE-ID CVE-2001-0144). The consequences of a truncation can be several. For example, if the *int* variable is used to allocate a buffer in the heap, a *buffer* overflow can later occur because less space than needed was allocated.

Notice that these problems can be very subtle. For instance, consider the case of a function that returns a pointer but in which the type of the return value is not defined [3]. In ILP32 this works perfectly. In LP64 the pointer is truncated to 32 bits. Truncations can also be caused by a cast, to give another example.

The tool To assess the existence of these vulnerabilities in open source applications written in C, we developed a tool called DEEEP (Detector of integEr vulnerabilitiEs in softwarE Portability, online at `http://deeep.homeunix.org/`). DEEEP is a *static source code analysis* tool, i.e., it analyzes source code without executing it. The tool does two forms of analysis and correlates their results to minimize the number of false positives (i.e, of lines of code tagged vulnerable that in fact are not). These two forms of analysis were not implemented from scratch, but instead two tools that did them already were used as building blocks:

- *Type checking* is the obvious mechanism to find vulnerabilities with integer manipulation because these are indeed caused by wrong usage of data types. Both building block tools were used to do type checking: Lint [5] and Splint [2]. Although Splint derives from Lint, a careful analysis of both has shown that they do not detect exactly the same integer manipulation bugs, there are omissions in both, so both have to be used.
- *Taint analysis* is a form of data flow analysis used to detect if tainted data reaches dangerous lines of code. The basic idea is to track the flow of data that comes from the program inputs to see if it reaches dangerous calls, like *memcpy* and others that are prone to buffer overflows. Taint analysis is done by Splint [2]. This tool is configured with data about (1) which parameters of library calls can not take tainted data (e.g., *malloc*'s only parameter), (2) which calls return tainted data (e.g., *gets*); and (3) how taintedness is propagated inside a program (e.g., tainted data operated with any data gives tainted data).

[*]This work was partially supported by the FCT through the Multiannual and the CMU-Portugal Programmes.

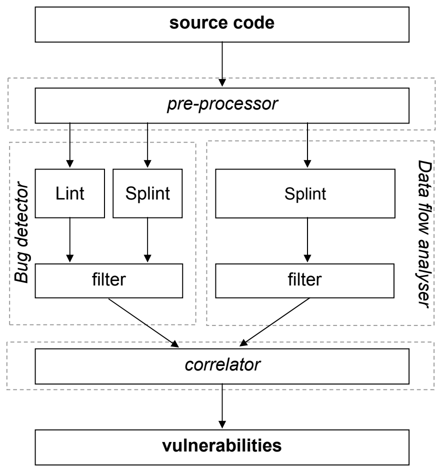

Figure 1. Architecture of the DEEEP tool.

These two forms of analysis play different roles. Type checking is in charge of finding bugs in lines of code than manipulate integers. However, these bugs are not necessarily vulnerabilities. For instance, they may not be used with tainted data (data that comes from the input) or reach parameters that have to receive untainted data. This is where taint analysis comes in: it tells which data flows reach dangerous lines. The tool has to pick these pieces of information and correlate them to say if there is a vulnerability. Nevertheless, false positives are still possible (see below).

Figure 1 shows the architecture of the tool. Some components are obvious, others need a short explanation. The pre-processor basically uses the makefiles of the program to run the cc/gcc pre-processor on all code files. The filters discard all alarms that are unrelated to specific vulnerabilities that are detected by DEEEP.

Experimental results The tool was first assessed with a set of reasonably short programs with vulnerabilities created for this purpose. The tool detected every single vulnerability. This assessment showed that the tool can indeed find the vulnerabilities that we are interested in. Then it was run with the code of 21 open source projects. The results are displayed in Table 1.

The results show that a large number of bugs with integer manipulation were found (column *integer warnings*). They were not only truncations but also overflows, underflows and signedess problems. The numbers of vulnerabilities flagged by the tool were much lower (column *vulnerabilities*) and all of them were false positives (next column). The manual analysis of each of these vulnerabilities has shown that they are not really vulnerabilities. There is indeed input data that is propagated to integer manipulation bugs and then to dangerous calls, but the logic of the program prevents these bugs from being attackable. In fact the main conclusion is that even this correlation of two forms of analysis is not enough to clearly distinguish bugs from vulnerabilities, eliminating false positives.

Application	Integer warnin.	Vuln.	False posit.	Files	Lines of code	Analysis time
wu-ftpd 2.6.2	217	0	-	50	22.629	25 sec
vsftpd 2.0.5	91	0	-	34	12.376	21 sec
sendmail 8.14.1	1.132	3	3	160	112.700	1:52 min
samba 3.0.26a	21.566	0	-	651	494.688	23:11 min
proftpd 1.3.0a	409	0	-	95	87.868	1:30 min
lighttpd 1.4.18	886	0	-	93	52.134	2:00 min
inetutils 1.5	980	0	-	175	79.793	1:16 min
dovecot 1.0.5	1.984	0	-	359	111.026	5:58 min
bind 9.4.1	1.298	0	-	604	323.860	4:24 min
asterisk 1.4.18	11.906	2	2	414	333.997	10:28 min
antiword 0.37	255	1	1	67	30.864	1:08 min
aircrack 0.9.3	353	4	4	14	21.147	0:35 sec
lsat 0.9.6	13	1	1	36	6.923	0:18 sec
ipac-ng 1.31	206	1	1	28	19.928	0:29 sec
rodmap 1.1.0	719	4	4	150	69.929	3:28 min
pen 0.17.2	95	1	1	5	3.772	0:21 sec
atop 1.22	169	3	3	14	12.030	0:32 sec
clamav 0.60	495	3	3	50	19.873	1:10 min
openldap 2.1	310	2	2	452	198.626	9:59 min
openafs 1.4.6	8.290	0	-	1221	689.920	15:25 min
wine 0.9.52	715	0	-	2197	1.833.251	1:57:38 h
Total	52089	25	25	6869	4537334	

Table 1. Results of running DEEEP with 21 open source projects. *Integer warnings* are the outcome of the *bug detector* component (after filtering). The column *false positives* was obtained through manual analysis off all vulnerabilities detected (previous column).

Final remarks This abstract briefly introduces the problem of vulnerabilities introduced when porting code from 32 to 64-bit CPUs. It presents a methodology based on correlating the results of two kinds of analysis to find these vulnerabilities, and a tool that implements that methodology, DEEEP. The tool was tested successfully with syntectic code with vulnerabilities introduced on purpose, but failed to find vulnerabilities in real applications with more than 4.5 million lines of code, a somewhat surprising negative result.

We warmly thank Per Mellstrand for introducing us to the problem of introducing vulnerabilities when porting code to 64-bit CPUs.

References

[1] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security*, Jan. 2007.

[2] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002.

[3] R. Mach. Moving to 64-bits. *Dr. Dobb's*, June 2005.

[4] Open Group. Data size neutrality and 64-bit support. In A. Josey, editor, *Go Solo 2 - The Authorized Guide to Version 2 of the Single UNIX Specification*. The Open Group, 1997. http://www.unix.org/version2/whatsnew/login_64bit.html.

[5] Sun Microsystems. C user's guide. http://docs.sun.com/source/806-3567/, 2000.