Christian Wressnegger*, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck

64-Bit Migration Vulnerabilities

DOI 10.1515/itit-2016-0041
Received August 19, 2016; accepted January 17, 2017

Abstract: The subtleties of correctly processing integers confronts developers with a multitude of pitfalls that frequently result in severe software vulnerabilities. Unfortunately, even code shown to be secure on one platform can be vulnerable on another, such that also the migration of code itself is a notable security challenge.
In this paper, we provide a high-level overview of integer-based vulnerabilities that originate in code which works as expected on 32-bit platforms but not on 64-bit platforms. The changed width of integer types and the increased amount of addressable memory introduce previously non-existent vulnerabilities that often lie dormant in existing software. To emphasize the lasting acuteness of this issue, we empirically evaluate the prevalence of these flaws in the scope of Debian stable ("Jessie") and 200 popular open-source projects hosted on GitHub.

Keywords: Software security, Data models, Integer-based vulnerabilities.

ACM CCS: Security and Privacy → Software and application security

1 Introduction

The migration of software from one platform to another may seem like a straight-forward task at first and, after over 10 years since 64-bit computing has reached the mass market, one might expect that technical obstacles introduced by 64-bit data models have long been resolved. Unfortunately, the reverse is true. Over the past years vulnerabilities solely induced by migration have been brought to light across wide-spread and well reviewed software projects, such as CVE-2005-1513 in qmail, CVE-2007-1884 in PHP, CVE-2013-0211 in libarchive or CVE-2014-9495 in libpng. On closer examination, the process of migrating program codes presents itself as far more involved than it appears at a first glance. While the LP64 data model changes widths of a few integer types only, these particular types are often used in the calculation of buffer sizes, offsets in memory, or the amounts of memory to copy from one location to another [2, 5] and thus are especially critical in this context. This is even aggravated by chains of type aliases (cf. the `typedef` keyword in C/C++) and signedness issues, which developers and even security experts often are not aware of.

Let us, as an example of a real vulnerability, consider the following simplified excerpt of a flaw in *zlib* version 1.2.8:

```
int len = attacker_controlled();
char *buffer = malloc((unsigned) len);
memcpy(buffer, src, len);
```

The functions `malloc` and `memcpy` define the number of bytes to allocate and the number of bytes to copy, respectively, as an integer of type `size_t`. Consequently, the variable `len` is implicitly cast to the unsigned type `size_t` in line 2 and 3 – such that this code is perfectly secure on all 32-bit platforms. However, if the code is compiled using the LP64 data model (e.g., used by 64-bit Linux) line 3 evokes a sign extension as `size_t` there is defined as being 64 bits wide. An attacker controlling the variable `len` can hence overflow the buffer by providing a negative number. For example, −1 is converted to `0x00000000ffffffff` in line 2 but to `0xffffffffffffffff` in line 3, resulting in a buffer overflow.

In this paper, we provide an overview of such *64-bit migration vulnerabilities* and summarize results presented in [13]. On the basis of different minimal working examples we demonstrate how (a) the changed widths of crucial integer types and (b) the larger address space that is available on 64-bit systems may lead to severe security flaws. To assess the prevalence of migration flaws in practice, we conduct an empirical analysis and search for such issues in the source code of 200 GitHub projects and all packages from Debian stable ("Jessie") marked as *Required*, *Important* or *Standard*. We find that integer truncations and signedness issues induced by 64-bit migration are abundant in both datasets. For example, `size_t` alone, which has a width of 64 bit under LP64, is truncated to 32-bit types in 78% of all Debian packages. Although the vast majority of these issues are not necessarily vulnerabilities,

***Corresponding author: Christian Wressnegger,** TU Braunschweig, Institute of System Security, D-38106 Braunschweig, Germany, e-mail: c.wressnegger@tu-bs.de
Fabian Yamaguchi, Alwin Maier, Konrad Rieck: TU Braunschweig, Institute of System Security, D-38106 Braunschweig, Germany

the sheer amount indicates that developers are unaware of the subtle changes resulting from migrating code to LP64.

The rest of this article is organized as follows: We systematically describe 64-bit migration vulnerabilities in Section 2. In Section 3 we present an empirical study on the prevalence of this unique class of vulnerabilities and discuss our findings in Section 4. Section 5 concludes the article.

2 Systemization

In the following, we first address different data models deployed over the years (Section 2.1) before we characterize different types of vulnerabilities that emerge when compiling code for a 64-bit data model that securely runs on 32-bit platforms. These vulnerabilities can be categorized by two generic sources of defects: changes in the width of integers (Section 2.2) and the larger address space available on 64-bit systems (Section 2.3).

2.1 Data models

A data model defines the width of integer types for a specific platform. Table 1 provides an overview of common data models used in the present and past, exemplary operating systems using them, as well as the number of bytes assigned to each type. For all models, the width of pointers and the `size_t` type correspond to the architectures' register size, e.g., IP16 and LLP64 specify the size of pointers as 2 byte and 8 byte, respectively.

The motivation behind the different definitions of basic integer types lies in preserving their relations as good as possible when migrating code between data models. Due to our focus on the transition from 32 bit to 64-bit, ILP32 serves as a reference point in this paper, as it is used on most 32-bit architectures. That is, we assume that a given program works as intended for ILP32 and look upon the differences when compiling the same program using a 64-bit data model.

If we compare ILP32 to LLP64 and LP64, as used by 64-bit Windows and most 64-bit Unix systems, respectively, we see that the type `int` is 32-bit wide for all three data models. While for ILP32 this means that `int` and pointers have the same width, on 64-bit data models `int` is only half as wide as the pointer type. The same holds true for the type `long` on LLP64. As a consequence, on both 64-bit data models an `int` variable can no longer be used to address the full range of memory. While there also exist other 64-bit data models, such as ILP64 and SILP64, these are only used on few platforms only and define the same width for `int`, `long` and pointers, which renders migrating code less problematic.

2.2 Effects of integer width changes

The difference in size of integers on 64-bit platforms in comparison to 32-bit data models introduces previously non-existent truncations and sign extensions in assignments. Surprisingly, the migration to 64 bit may even flip the signedness of comparisons and render checks for buffer overflows ineffective. In the following, we discuss each of these problems in detail.

2.2.1 New truncations

A truncation occurs when an expression is assigned to a type narrower than that of the expression itself. Particularly noteworthy are those assignments that behave differently between ILP32 and LLP64 or LP64, such as conversions from `size_t` to `unsigned int` or `long` to `int`. In these cases, new truncations occur that are specific to the migration process from 32-bit to 64-bit data models. In addition to these simple truncations, the migration of the data model additionally introduces two vulnerability patterns related to the handling of pointers.

Table 1: Widths of basic integer types in bytes for different data models and examplary operating systems making use of these [6, 9].

data model / data type	IP16 (PDP-11 Unix)	IP16L32 (PDP-11 Unix)	LP32 (Win16)	ILP32 (Win32, Linux)	LLP64 (Win64)	LP64 (Linux)	ILP64 (HAL)	SILP64 (UNICOS)
`pointer/size_t`	2	2	4	4	8	8	8	8
`short`	–	2	2	2	2	2	2	8
`int`	2	2	2	4	4	4	8	8
`long`	–	4	4	4	4	8	8	8
`long long`	–	–	8	8	8	8	8	8

Incorrect pointer differences. The length of a memory region can be determined by subtracting pointers, returning an integer of type `ptrdiff_t`, which has the same width as a pointer. Unfortunately, it is common practice to store such differences in a variable of type `int`. This is unproblematic on all 32-bit platforms, since both types are of the same size. However, on LP64 and LLP64 `ptrdiff_t` is 64-bit wide, while the width of `int` remains unchanged, which means that the difference is truncated to 32 bit and thus may cause loss of information. Listing 1 shows an exemplary vulnerability of this type. Compiling the code produces no warnings, yet on 64-bit platforms, line 5 introduces an integer truncation. The example shows a typical pattern for processing an input string `str` line-by-line and determining a line's length by the difference of end and start pointers. If one input line exceeds 4 Gigabyte in length, the variable `len` only stores the truncated length as it is 32 bit wide. For instance, if `MAX_LINE_SIZE = 100` and `eol - str = 0x1000000ff`, `len` is truncated to `0x000000ff` and finally triggers a buffer overflow in line 8.

```
1 char buf[MAX_LINE_SIZE];
2 char *eol = strchr(str, '\n');
3 *eol = '\0';
4
5 unsigned int len = eol - str;
6 if(len >= MAX_LINE_SIZE)
7          return -1;
8 strcpy(buf, str);
```

Listing 1: Example of a 64-bit migration vulnerability caused by incorrect pointer differences.

Unfortunately, vulnerabilities of this type are supported by the design of standard library functions, such as `fgets`, `fseek` and `snprintf`, which receive or return size information as type `int` and `long`. The common idiom of using variables of type `int` to iterate over buffers further adds to this problem (see Section 2.3.1).

Casting pointers to integers. Closely related are casts from pointers to integers. While this programming pattern is generally discouraged, casting pointers to `int` is unproblematic on all 32-bit platforms, as pointers and integers have the same size. In contrast, on LP64 and LLP64 where pointers are 64-bit wide this practice leads to *latent pointer truncations* [10]. These truncations are latent in the sense that they go unnoticed as long as the pointers refer to locations within the first 4 Gigabyte of the address space. For these pointers, a truncation does not change their value as only preceding zeros are removed. Attackers, however, may purposely increase the amount of memory allocated by the program to ensure that pointers outside this safe range are created. Still, these vulnerabilities are rather rare and a successful exploitation is rendered difficult by address space layout randomization (ASLR) [1].

2.2.2 New signedness issues

Two types of integer signedness issues arise as code is ported from 32-bit to 64-bit platforms. First, sign extensions may occur as signed integers are converted to unsigned types that have become wider than their ILP32 equivalents. Second, the signedness of comparisons potentially changes, rendering checks to protect from buffer overflows ineffective.

Sign extensions. When converting from one signed type to another wider *signed* type, a sign extension is performed for value preservation. Converting a signed type to a wider *unsigned* type follows the same principle, but the resulting value is eventually interpreted as unsigned integer. In effect, negative numbers are converted into large positive numbers, a possible source for vulnerabilities. For the LLP64 data model, new sign extensions occur for conversions from `int` and `long` to `size_t` and for LP64 from `int` to `unsigned long`and `size_t`. From a security perspective, conversions to the `size_t` type appear to be especially fruitful when looking for vulnerabilities as the example of the *zlib* vulnerability presented in the introduction demonstrates.

Signedness of comparisons. Checks to ensure that a buffer does not overflow are only effective if they correctly account for the signedness of the integers involved. Typically, this means that all integers should be converted to unsigned types prior to comparison. In many cases, explicit conversions can be omitted on 32-bit systems as integer conversion rules ensure that the comparisons will be performed unsigned. This, however, is not guaranteed on 64-bit platforms anymore, bringing forth comparisons that change their signedness when being ported. For instance, a comparison involving `long` and `unsigned int` is *unsigned* on both, ILP32 and LLP64, but *signed* on the LP64 data model.

Listing 2 presents a corresponding vulnerability. An attacker-controlled value is first stored in a `long` integer named `len` on line 2, and then checked to ensure it does not exceed the buffer size `BUF_SIZE` on line 4. Finally, `len` bytes are copied into the buffer. As in the previous example, compiling this code produces

```
1 const unsigned int BUF_SIZE = 128;
2 long len = attacker_controlled();
3
4 if(len > BUF_SIZE)
5        return;
6 memcpy(buffer, src, len);
```

Listing 2: A check to avoid buffer overflows on 32-bit systems that is ineffective on LP64 platforms.

no warnings. Moreover, the comparison between `len` and `BUF_SIZE` is unsigned on 32-bit data models. This is the case because `long` and `unsigned int` have the same width and therefore `long` cannot hold the full range of `unsigned int`. Consequently, `len` gets reinterpreted as unsigned value to conduct the comparison. For instance, given `len = -1` the comparison is performed unsigned as `0xffffffff > 0x00000080`. Although a reinterpretation of the value occurs, the result still matches the developer's expectations.

In contrast, on LP64 the type `long` is 8 bytes wide, while an `unsigned int` is only 4 bytes wide. Therefore, a variable of type `long` can hold the full range of an `unsigned int`, and a signed comparison is performed. This is problematic, as the check in line 4 can be bypassed by supplying a negative value, for instance `-1`, for `len`. When copying data on line 6, this value is sign-extended and interpreted as unsigned integer, `0xffffffffffffffff`, resulting in a buffer overflow.

2.3 Effects of a larger address space

In addition to flaws that result from changes in integer widths, code running on 64-bit platforms has to be able to deal with larger amounts of memory as the size of the address space has increased from 4 Gigabytes to several hundreds of Terabytes. In effect, the developer can no longer assume that buffers larger than 4 Gigabytes *cannot* exist in memory. As a result, additional integer truncations and overflows emerge, which do exist on 32-bit data models in the first place, but cannot be triggered on the corresponding platforms in practice.

2.3.1 Dormant integer overflows

A security-relevant integer overflow cannot be detected by reasoning about the types of variables alone. Instead, the range in which these variables operate also needs to be considered. A larger address space allows (a) larger objects to be created and (b) a larger number of objects to be used. Thus, code that performs arithmetic operations on the sizes or number of objects with variables narrower than that of pointers become candidates for integer overflows on 64-bit platforms.

```
1 unsigned int i;
2 size_t len = attacker_controlled();
3 char *buf = malloc(len);
4
5 for(i = 0; i < len; i++) {
6         *buf++ = get_next_byte();
7 }
```

Listing 3: Buffer overflow resulting from an integer overflow due to larger strings on 64-bit platforms.

Listing 3 provides an example of an integer overflow resulting from large objects, which also does not trigger a compiler warning. For LP64 and LLP64, the type `unsigned int` is narrower than `size_t`. Thus, if the attacker-controlled value `len` is larger than `UINT_MAX`, the loop-variable `i` can never attain a value greater or equal to `len` as it would first overflow and eventually result in a loop that endlessly copies data into the buffer. Platforms using ILP32, however, are not affected since `SIZE_MAX` equals `UINT_MAX` – in other words, the loop terminates before `i` overflows.

2.3.2 Dormant signedness issues

In addition to truncations, signedness issues may also lie dormant in existing code and become exploitable as the size of the address space grows. A common occurrence of such dormant signedness issues is the practice of assigning the result of `strlen` to a variable of type `int`. For strings longer than `INT_MAX`, this results in a negative length. However, on 32-bit platforms, exploiting this type of flaw is deemed unrealistic due to the restricted amount of memory available [5, Chp. 18 pp. 494]. On 64-bit platforms, however, strings of this size can be easily allocated by a single process, making it possible to trigger these dormant signedness issues.

Listing 4 shows a corresponding vulnerability. The length of the attacker-controlled string is determined using `strlen` and is assigned to a variable of type `int`. If the attacker controlled input is larger than `INT_MAX` but smaller than `UINT_MAX`, the value stored in `len` is mistakenly interpreted as a negative number and the check in line 4 is rendered ineffective. As `len` is subsequently

passed to `memcpy`, it is sign-extended and interpreted as `unsigned int`, causing a buffer overflow in line 6.

```
1 char buffer[128];
2 int len = strlen(attacker_str);
3
4 if(len >= 128)
5         return;
6 memcpy(buffer, attacker_str, len);
```

Listing 4: Buffer overflow caused by the common pattern of assigning the result of `strlen` to an `int`.

2.3.3 Unexpected behavior of library functions

Several standard C library functions have been originally designed with 32-bit data models in mind and thus become vulnerable to truncations, overflows or signedness issues. Although some of these functions have been adapted to 64-bit data models, developers are often not aware of the changed functionality.

String formatting. Functions for printing strings, such as `fprintf`, `snprintf` and `vsnprintf` have been designed with the assumption that strings cannot be longer than `INT_MAX`. While this assumption is reasonable on 32-bit platforms, it does not hold true for 64-bit data models. Let us, as an example, consider `snprintf`, which writes a string to a buffer `s` according to a format string `fmt`.

```
int snprintf(char *s,
             size_t n, const char *fmt, ...)
```

The function copies at most `n` bytes and returns the number of bytes that *would have been* written. On 64-bit platforms the expanded format string, however, may be larger than `INT_MAX`, making it impossible to return its size as an `int`. In this case the C99 standard demands that `snprintf` returns a fixed value of −1 [4, Sec. 7.19.6]. In practice, this can result in vulnerabilities when programmers directly make use of the return value to shift pointers. Listing 5 exemplarily shows a vulnerable implementation of a log function that writes messages to a global buffer of `BUF_LEN + 1` bytes in size.

The log function returns −1 once the return value of `snprintf` has exceeded the overall size of the buffer (line 7–9). Specifying an input string longer than `INT_MAX`, which is easily possible on 64-bit platforms, results in `snprintf` returning −1 on line 5 – irrespective of the maximal number of bytes allowed to write. This bypasses the check on line 7 and subtracts from the index variable `pos`, causing it to underflow. A subsequent call to `log` then corrupts the stack memory.

```
 1 int pos = 0;
 2 char buf[BUF_LEN+1];
 3
 4 int log(char *str) {
 5    int n = snprintf(buf+pos, BUF_LEN-pos, str);
 6
 7    if(n > BUF_LEN-pos) {
 8         pos = BUF_LEN;
 9         return -1;
10    }
11    return (pos += n);
12 }
```

Listing 5: Stack-corruption vulnerability on 64-bit systems due to unexpected behavior of `snprintf`.

File processing. Similar to the `printf` family of functions, the standard C library functions for processing files, such as `ftell`, `fseek` and `fgetpos`, are not designed to deal with the effects of 64-bit integer numbers, particularly, files larger than 4 Gigabyte. This problem is well known and is addressed by the introduction of 64-bit aware counterparts, `ftello`, `ftello64` or `__ftelli64`. However, our empirical study shows that `ftell` still is widely used instead of the better alternatives (see Section 3). Furthermore, the function `ftell` exhibits an undocumented behavior when confronted with large files. It is specified to return the current position of a file pointer as value of type `long`, which is 32 bit wide on platforms using the LLP64 data model. While the C99 standard specifies a return value of −1 for failures [4, Sec. 7.19.3], the *Microsoft Visual C++ Runtime Library*'s implementation returns 0 if the current position exceeds `LONG_MAX` (`0xffffffff`), which gives rise to security problems.

Listing 6 shows an exemplary vulnerability in a piece of code that reads a file of hexadecimal values encoded in textual form (e.g., `30 31 32 ...`) and stores it as decoded bytes in a buffer `buf`. To this end, the code first determines the file's size by seeking to its end and obtaining the file position using `ftell` (line 4–6). Finally, the byte values are written to the buffer by iteratively calling `fscanf` until `EOF` is reached (lines 10–12). On Microsoft Windows 64-bit a vulnerability can be triggered using files larger than 4 Gigabytes, as the call to `ftell` returns zero and only one byte is allocated for the buffer (line 8). In effect, the copy loop corrupts the heap by writing the complete file to memory not allocated by the process.

```
 1 int i;
 2 char *buf;
 3
 4 FILE* const f = fopen(filename, "r");
 5 fseek(f, 0, SEEK_END);
 6 const long size = ftell(f);
 7
 8 buf = malloc(size / 2 + 1);
 9
10 fseek(f, 0, SEEK_SET);
11 for (; fscanf(f, "%02x", &i) != EOF; buf++)
12         *buf = i;
```

Listing 6: Buffer overflow for files larger than `UINT_MAX` caused by unexpected return value of `ftell`.

3 Empirical study

We proceed to analyze how wide-spread 64-bit migration issues are in today's software. To this end, we conduct two empirical experiments. First, we assess the prevalence of problematic type conversions in general, considering all *implicit conversions* that may alter a value during assignments or in expressions (Section 3.1). Second, we refine our search and automatically look for *programming patterns* that are characteristic for 64-bit migration flaws (Section 3.2).

3.1 Implicit type conversions

In this experiment we study how often type conversions potentially go wrong. To this end, we inspect all 198 source packages from Debian stable ("Jessie", release 8.2) tagged as either *Required*, *Important* or *Standard* and are written in the C/C++ programming languages. We compile each package on Debian 32-bit and Debian 64-bit and inspect all warnings raised.

On request, GCC, LLVM's clang, and other compilers emit warnings when an assignment, arithmetic operation or a comparison is applied to operands of incompatible integer types and an implicit conversion is required. Frequently, these compiler flags are however *not* used due to the sheer amount of warnings potentially raised in practice [7]. As a matter of fact, we find that none of the 198 Debian packages uses one of these flags. For our study we hence explicitly add: `-Wconversion` for width conversion, `-Wsign-conversion` for changes in signedness, `-Wsign-compare` for comparisons of signed and unsigned types and `-Wfloat-conversion` for conversions that involve a loss in floating point precision. Table 2 summarizes the results. For each conversion type we list the total count of warnings raised by the compiler on the 64-bit system per package and especially highlight warnings that have emerged from the migration process. We find that the vast majority of warnings are width and sign conversions with 442 and 259 warnings per package, respectively. Especially, the conversion from `size_t` to `int` and vice versa appears to be problematic in practice, spawning 21 527 warnings in core packages of Debian stable. All these warnings are exclusive to 64 bit and do not occur on 32-bit platforms. By contrast, sign comparisons only slightly increase due to the 64-bit migration. However, in line with the examples given in Section 2.2.2 migration vulnerabilities often occur on 64-bit platforms due to comparisons that remain signed rather than being implicitly converted to unsigned. Hence, the amount of warnings resolved in comparison to a 32-bit platform has to be taken into account as well, such that in total 15% of the warnings can be considered critical.

3.2 Patterns of 64-bit migration issues

Of course, not all implicit conversions indicate a bug or even a vulnerability. We hence narrow down this vast amount of suspect locations by specifically looking for

Table 2: Number of implicit type conversions per package on 64 bit. The first value denotes all warnings raised, the value in brackets the amount that is *exclusive* to 64 bit and that does not occur on 32-bit systems.

Debian stable		Average per package			
Category	# packages	-Wconversion	-Wsign-conversion	-Wsign-compare	-Wfloat-conversion
Required	53	576 (334)	1009 (216)	18 (2)	5 (1)
Important	56	738 (437)	976 (269)	33 (1)	10 (0)
Standard	89	913 (510)	993 (279)	28 (1)	3 (1)
*	198	773 (442)	993 (259)	27 (1)	5 (1)

Table 3: Number of specific patterns for 64-bit migration issues in source packages of Debian stable ("Jessie", release 8.2) and 200 popular C/C++ projects hosted on GitHub, relative to their absolute usage.

Code-base	P1: `atol`	P2: `memcpy`	P3: `loops`	P4: `strlen`	P5a: `snprintf`	P5b: `ftell`
Debian Jessie	21.49% (133)	7.76% (2536)	8.47% (1264)	13.85% (7595)	27.55% (762)	64.74% (628)
GitHub	18.66% (25)	15.19% (2918)	12.56% (658)	22.54% (3572)	34.79% (502)	85.05% (182)
Average	20.98% (158)	10.51% (5454)	9.53% (1922)	15.80% (11 167)	30.03% (1264)	68.41% (810)

code patterns that may cause unintended operations on 64-bit platforms. To this end, we make use of techniques from control-flow and data-flow analysis to model specific patterns of 64-bit migration issues. In particular, we characterize patterns from the 5 categories presented in Section 2 on the basis of practical examples and count the occurrences of these in two code bases: we again consider the packages from Debian stable described in the previous section and additionally examine the 200 (at the time of writing) most popular C/C++ projects on GitHub. Table 3 summarizes our findings.

21% of all calls to function `atol` are assigned to a variable of type `int` instead of `long`, causing a truncation on 64-bit systems (P1). Also, developers frequently pass signed integers of type `int` to function parameters defined as `size_t`. In case of the `memcpy` function and its parameter for specifying the number of bytes to copy, roughly 10% of the calls are used incorrectly, allowing for the malicious use of implicit sign-extensions (P2). Our pattern modeling integer overflows induced by simple `for` loops reveals that 9.5% increment an `int` variable although the loop-counter is specified as `size_t` (P3). 15% of all calls to `strlen` are falsely assigned to a variable of type `int` rather than `size_t` (P4). Finally, the `snprintf` and `ftell` functions are incorrectly used and their results insufficiently checked in 30% and 70% of all cases, respectively (P5a & P5b). For more details on these results and the methodology used please refer to [13].

In summary we observe that projects included in Debian appear to exhibit less such patterns for 64-bit migration flaws than the projects retrieved from GitHub – the absolute number however suggests a significant potential for misuse.

4 Discussion

Ideally, flaws induced by migrating from 32-bit to 64-bit platforms are addressed by thorough code audits that specifically focus on problematic type conversions and related code patterns. Our study however suggests that this currently is not put into practice effectively and shows that vulnerabilities resulting from the migration process are still a major issue.

While a thorough understanding of the underlying issue, as provided by this and similar articles [2, 8, 9, 12, 13], and the awareness for this particular aspect of software security is a key stepping stone, we believe that the current prevalence of such flaws reflects the lack of tools assisting the detection but also the development process.

Improved error reporting. As demonstrated in Section 3.1 even well-reviewed code from mature projects contains a multitude of type-conversion warnings. C/C++ projects from Debian stable tagged as *Required*, *Important* or *Standard* spawn 1798 warnings related to different kinds of conversions on average, 703 of which are exclusive to the migration to 64-bit platforms. Whether or not these express actual flaws or even security issues is unclear. It, however, appears that those that actually are security flaws, get lost in the sheer amount of warnings. Presumably for this exact reason, none of the inspected Debian packages makes use of the `-Wconversion` flag and in doing so, turns a blind eye on these issues.

Factoring out the functionality of GCC's `-Wconversion` flag that concerns data types that have changed in size due to migration to 64-bit platforms to a separate flag as deployed in IBM's XL compiler [3], for instance, is a valuable first step. Such an additional flag can then be issued individually or automatically set on specifying `-Wconversion` to preserve the current functionality of the compiler. Although, this already reduces the amount of warnings by 60%, the absolute number of warnings in complex software projects might still be too large to be handled at once.

Making use of data that arise from program analysis already employed by compilers, can be used to restrict warnings to more specific situations, as for instance, the lack of some sort of check on values for which a conversion warning is issued. In case of the examined Debian packages this would reduce the number of warnings by additional 95% to merely 30 instances. However, such analy-

ses come at a computational cost that often does not fit the requirements of a performance-oriented compiler framework.

5 Conclusion

In this article, we categorize and review vulnerabilities made possible by the migration to 64-bit platforms. We provide a high-level reference as well as minimal working examples for practitioners and show the prevalence of such vulnerabilities in mature and well-tested software. For example, to a large extend developers appear to unjustifiably treat the unsigned type `size_t` and (`unsigned`) `int` as equal, leading to in total 21 527 warnings in Debian stable. Moreover, we look for particular patterns of 64-bit migration flaws to refine our findings on implicit type conversions. For instance, 10% of all invocations to the `memcpy` function in the inspected Debian and GitHub projects, are called with a signed value of 32-bit in size rather than the 64 bit wide `size_t` as parameter for the number of bytes to copy. Based on our findings we additional motivate further directions of research to better address 64-bit migration vulnerabilities.

Acknowledgement: The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the projects APT-Sweeper (FKZ 16KIS0307) and INDI (FKZ 16KIS0154K).

References

1. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. of USENIX Security Symposium*, 2003.
2. W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proc. of International Conference on Software Engineering(ICSE)*, 2012.
3. IBM Corp. XL C/C++: Optimization and programming guide. Technical report, IBM Corp., 2012.
4. ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
5. J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.
6. T. Lauer. *Porting to Win32™: A Guide to Making Your Applications Ready for the 32-Bit Future of Windows™*. Springer, 1996.
7. M. López-Ibáñez and I. L. Taylor. The new Wconversion option. https://gcc.gnu.org/wiki/NewWconversion, visited December 2016.
8. R. Mach. Moving to 64-bits. *C/C++ Users Journal*, 5, 2005.
9. J. R. Mashey. The long road to 64 bits. *ACM Queue Magazine*, 4 (8):24–35, 1996.
10. Microsoft Security Research and Defense. Software defense: mitigating common exploitation techniques, 2013.
11. The Open Group. 64-bit and data size neutrality. http://www.unix.org/version2/whatsnew/lp64_wp.html, 2000.
12. Viva64. Lessons on development of 64-bit c/c++ applications. http://www.viva64.com/en/l/full/, visited December 2016.
13. C. Wressnegger, F. Yamaguchi, A. Maier, and K. Rieck. Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

Bionotes

Christian Wressnegger
TU Braunschweig, Institute of System Security, D-38106 Braunschweig, Germany
c.wressnegger@tu-bs.de

Christian Wressnegger is a research associate and PhD candidate at the Institute of System Security at the Technische Universität Braunschweig. Prior to taking this position, he has been working for several institutions both in academia and industry in various fields of computer security and machine learning. He graduated with a Masters degree in Computer Science from Technische Universität Graz in 2008. His research interests revolve around the detection and prevention of malware, vulnerability discovery, and applied machine learning.

Dr. Fabian Yamaguchi
TU Braunschweig, Institute of System Security, D-38106 Braunschweig, Germany
f.yamaguchi@tu-bs.de

Fabian Yamaguchi is a post-doctoral researcher at the Institute of System Security of the Technische Universität Braunschweig. He received his Diploma in Computer Engineering from Technische Universität Berlin in 2011 and his Doctorate in Computer Science from the University of Göttingen in 2015. He was awarded the CAST/GI Dissertation prize for his thesis entitled Pattern-based Vulnerability Discovery and received the German Prize for IT-Security in 2016. In his research, he focuses on program analysis, vulnerability discovery, machine learning, and de-anonymization.

Alwin Maier
TU Braunschweig, Institute of System Security, D-38106 Braunschweig, Germany
alwin.maier@tu-bs.de

Alwin Maier is a research associate and PhD candidate at the Institute of System Security of the Technische Universität Braunschweig. Prior to that position, he worked at the University of Göttingen where he also graduate with a Masters Degree in Applied Computer Science in 2015. His research interests include various aspects of computer security and machine learning with a focus on program analysis and its applications in computer security.

Prof. Dr. Konrad Rieck
TU Braunschweig, Institute of System Security, D-38106 Braunschweig, Germany
k.rieck@tu-bs.de

Konrad Rieck is a Professor at Technische Universität Braunschweig, where he leads the Institute of System Security. Prior to this, he has been working at the University of Göttingen, Technische Universität Berlin and Fraunhofer Institute FIRST. He graduated in 2004 and received a Doctorate from Technische Universität Berlin in 2009. Konrad Rieck is a recipient of the CAST/GI Dissertation Award, the Google Faculty Research Award and the German Prize for IT-Security. His interests revolve around computer security and machine learning, including the detection of computer attacks, the analysis of malicious code, and the discovery of vulnerabilities.