Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms

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

Institute of System Security
TU Braunschweig

Abstract

Subtle flaws in integer computations are a prime source for exploitable vulnerabilities in system code. Unfortunately, even code shown to be secure on one platform can be vulnerable on another, making the migration of code a notable security challenge. In this paper, we provide the first study on how code that works as expected on 32-bit platforms can become vulnerable on 64-bit platforms. To this end, we systematically review the effects of data model changes between platforms. We find that the larger width of integer types and the increased amount of addressable memory introduce previously non-existent vulnerabilities that often lie dormant in program code. We empirically evaluate the prevalence of these flaws on the source code of Debian stable ("Jessie") and 200 popular open-source projects hosted on GitHub. Moreover, we discuss 64-bit migration vulnerabilities that have been discovered as part of our study, including vulnerabilities in Chromium, the Boost C++ Libraries, libarchive, the Linux Kernel, and zlib.

Keywords

Software security; Data models; Integer-based vulnerabilities

1. INTRODUCTION

64-bit CPU architectures have become the main platform for server and desktop systems. While 64-bit computing has been used in research systems for almost four decades, it took until 2003 for the underlying architectures to reach the mass market. Since then, all major operating systems have been ported to support 64-bit architectures, including Linux, Windows and OS X. Software running on these systems benefits from a huge address space that enables operating with Gigabytes of memory and provides the basis for memory-demanding computations.

The migration of software from one to another platform may seem like a straight-forward task and, after over 10 years, one might expect that technical obstacles introduced by 64-bit data models have long been resolved. However, this migration is far more involved than it seems, as it induces several subtle differences in the resulting code. For example, the LP64 data model seemingly affects a few integer types only, yet these changes unleash an avalanche of new type aliases (`typedef`) and signedness issues, which developers and even security experts are not aware of. As a result, 64-bit issues are rather the rule than the exception in migrated code and there exist several examples of vulnerabilities solely induced by migration, such as CVE-2005-1513 in qmail, CVE-2007-1884 in PHP, CVE-2013-0211 in libarchive and CVE-2014-9495 in libpng.

In this paper, we provide the first systematic study of these *64-bit migration vulnerabilities*. To this end, we analyze common 64-bit data models for C/C++ and identify insecure programming patterns, when porting code from 32-bit architectures. We determine two interdependent sources for such vulnerabilities: (a) the changed integer widths and (b) the very large address space that allows to allocate massive amounts of memory. For each of these sources, we analyze the underlying root causes and pinpoint necessary conditions for their occurrence, thereby providing insights into migration vulnerabilities as well as a reference for developers. Although there exists a large body of work on integer-based flaws [e.g., 2, 6, 9, 24, 30, 34, 42, 43, 45], to the best of our knowledge, this is the first study that investigates vulnerabilities induced only by 64-bit migration.

To assess the presence of migration flaws in practice, we conduct an empirical analysis and search for 64-bit 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, the unsigned type `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, the sheer amount indicates that developers are unaware of the subtle changes resulting from migrating code to LP64.

Finally, we exemplify the security risk of 64-bit migration by presenting case studies on vulnerabilities that have been discovered as part of our study. For each of the identified classes of 64-bit issues, we present a corresponding vulnerability in a high-quality software project, including Google's *Chromium*, the *GNU C Library*, the *Linux Kernel* and the *Boost C++ Libraries*. Our analysis reveals that migration vulnerabilities are a widespread problem that is technically difficult to solve and requires significant attention during software development and auditing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'16, October 24 - 28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978403

Let us, as one example of a real vulnerability, consider the following simplified excerpt of a flaw discovered during our study in *zlib* version 1.2.8 (see Section 5 for more details):

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

This code is perfectly secure on all 32-bit platforms, as the variable `len` is implicitly cast to `size_t` in line 2 and 3 when passed to `malloc` and `memcpy`. However, if the code is compiled using the LP64 data model, line 2 and 3 produce different results, where a 64-bit sign extension is performed in line 3. An attacker controlling the variable `len` can thus overflow the buffer by providing a negative number. For instance, -1 is converted to `0x00000000ffffffff` in line 2 and `0xffffffffffffffff` in line 3, resulting in a buffer overflow.

In summary we make the following contributions:

- **64-bit migration vulnerabilities.** We systematically study and define vulnerabilities induced by migrating C/C++ program code to 64-bit data models.
- **Empirical study.** We present an extensive study that highlights the prevalence of 64-bit issues in mature, well-tested software such as Debian stable.
- **Practical case-studies.** We discuss 64-bit vulnerabilities, discovered by us based on this systemization, in high-quality software for *all* presented classes of flaws.

The rest of this paper is organized as follows: We review integer-related issues in Section 2 and systematically define 64-bit migration vulnerabilities in Section 3. An empirical study and case studies on these vulnerabilities are then presented in Section 4 and Section 5, respectively. We discuss countermeasures in Section 6 and related work in Section 7. Section 8 concludes the paper.

2. SECURITY OF INTEGER TYPES

Many software vulnerabilities are rooted in subtleties of correctly processing integers, in particular, if these integers determine the size of memory buffers or locations in memory. Leveraging these flaws, an attacker can trigger buffer overflows, write to selected memory locations or even execute arbitrary code. In this section we provide background information on the security of integers in C/C++ and focus on the effect the data model has on the existence of vulnerabilities. We begin by providing information on integers (Section 2.1), and continue to discuss the most prominent data models implementing them. Finally, we describe the resulting common classes of integer-related vulnerabilities (Section 2.3).

2.1 Integer Types in C

Conversions between integer types provide the basis for integer-related vulnerabilities, in particular, for those that result when porting code to 64-bit architectures. The C standard defines five base types for these integers, namely, `char`, `short`, `int`, `long`, and `long long` [18, Sec. 6.2.5].

Each of these integer types exists in two variants: an unsigned type to store positive numbers and a corresponding signed type that can also hold negative numbers. Moreover, each type is associated with a unique natural number known as its conversion rank. While the C standard does not explicitly define these ranks, it specifies an ordering among them. Consequently, we define the following platform-independent properties for each integer type T:

- *Signedness.* We denote the signedness of an integer type T by $S(T) \in \{0, 1\}$, where $S(T) = 0$ corresponds to unsigned and $S(T) = 1$ to signed types.
- *Conversion rank.* We denote the rank of an integer type T by $R(T) \in \mathbb{N}$, where $R(\texttt{char}) < R(\texttt{short}) < R(\texttt{int}) < R(\texttt{long}) < R(\texttt{long long})$.

The conversion rank orders integers by size, but does not specify the exact number of bits occupied by the different types. This gap is filled by the platform-dependent *data model*, which associates each type T with a concrete *width* $W(T) \in \{1, 2, 4, 8\}$. Just like the rank, the width of the signed and unsigned versions of a basic data type are required to be equal. In combination with the integer's signedness, the width of an integer specifies the *range* I of numbers that it can represent:

$$I(T) = \begin{cases} [0, 2^{8 \cdot W(T)} - 1] & \text{if } S(T) = 0 \\ [-2^{8 \cdot W(T)-1}, 2^{8 \cdot W(T)-1} - 1] & \text{otherwise} \end{cases}$$

Identifying sources of vulnerabilities as code is ported to a 64-bit data model ultimately requires the integer width to be taken into account. However, it should be noted that the width of a data type with lower rank must always be lower or equal to that of types with higher rank. That is, for any two types T_1 and T_2 with $R(T_1) < R(T_1)$ holds $W(T_1) \leq W(T_2)$. This observation is of great value for systematically identifying problems when code is ported to 64-bit platforms (Section 3).

2.2 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 [40], 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 focus on 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 the 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 and, moreover, use the same width for `int`, `long` and pointers, which renders migrating code less problematic.

data model	IP16	IP16L32	LP32	ILP32	LLP64	LP64	ILP64	SILP64
data type	(PDP-11 Unix)		(Win16)	(Win32, Linux)	(Win64)	(Linux)	(HAL)	(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

Table 1: Widths of basic integer types in bytes for different data models and operating systems [21, 27].

2.3 Integer-Related Vulnerabilities

Several vulnerabilities are rooted in subtle flaws as integers are processed to calculate the size of buffers, offsets in memory, or amounts of memory to copy from one location to another [see 9, 20]. Three common sources for these integer-related vulnerabilities exist: truncations, underflows/overflows, and signedness issues [6]. In the following, we illustrate each of these flaws by example where we assume an ILP32, LP64, or LLP64 data model. Moreover, we provide working definitions for each type of flaw, which we make use of in the rest of the paper.

2.3.1 Integer Truncations

For an arbitrary assignment $x = e$, where x is a variable of type $\langle x \rangle$, and e is an expression of type $\langle e \rangle$, a truncation occurs when $W(\langle x \rangle) < W(\langle e \rangle)$, that is, the width of the target variable x is smaller than that of the expression e it is asked to store. Neither rank nor signedness of the integers tell us whether a truncation occurs. Instead, the existence of truncations is entirely dependent on the width of integers, and therefore, on the data model. However, we know that the width of $\langle x \rangle$ can only be smaller than that of $\langle e \rangle$, if the rank $R(\langle x \rangle)$ is smaller than the rank $R(\langle e \rangle)$ and therefore, we can focus on these cases when examining possible truncations. It is important to note that x may be an implicit variable not directly visible in the code, for instance, when e is directly assigned to a function parameter or used in a condition.

```
1  unsigned int x = attacker_controlled();
2  unsigned short y = x;
3  char *buffer = malloc(y);
4  memcpy(buffer, src, x);
```

Figure 1: Integer truncation → buffer overflow.

Figure 1 shows an example of an integer truncation that leads to a buffer overflow. An attacker-controlled value is read and stored in the integer `x` of type `unsigned int` (line 1). Subsequently, `x` is assigned to the variable `y` of type `unsigned short` (line 2). Finally, a buffer of size `y` is allocated (line 3), and `x` bytes are copied into this buffer (line 4). The problem with this code is that for all three data models, integers of type `int` are wider than integers of type `short`, and therefore, the value of `x` is truncated before it is assigned to `y` by discarding leading bits until the width of `y` is met. In effect, if the attacker chooses a sufficiently large integer, e.g., `x = 0xffffffff`, the amount of data copied into the buffer is larger than its allocated size `0x0000ffff`.

2.3.2 Integer Overflows

For an arbitrary expression $e_1 \circ e_2$, an integer overflow or underflow occurs if the result obtained by evaluating the expression $e_1 \circ e_2$ does not fall into the range $I(\langle e_1 \circ e_2 \rangle)$. The existence of overflows therefore depends on the arithmetic operation $\circ$ as well as all the results obtained by evaluating the sub-expressions e_1 and e_2.

```
1  unsigned int x = attacker_controlled();
2  char *buffer = malloc(x + CONST);
3  memcpy(buffer, src, x);
```

Figure 2: Integer overflow → buffer overflow.

Figure 2 illustrates a buffer overflow that is triggered by an integer overflow. Similar to the previous example, the attacker controls a variable named `x`, which is of type `unsigned int`. A buffer of size `x` plus a constant is subsequently allocated, and `x` bytes are copied into the buffer. Unfortunately, the value obtained by adding a constant to `x` may be outside the range of the type `unsigned int`, for instance, `0xffffffff + 0x100`. In effect, the addition wraps around, resulting in a buffer size smaller than `x`, `0x000000ff`. A subsequent copy operation then writes `0xffffffff` bytes into the too small buffer.

2.3.3 Integer Signedness Issues

Finally, for an arbitrary assignment $x = e$, a change in signedness only occurs when $S(\langle x \rangle) \neq S(\langle e \rangle)$, that is, the signedness of the variable x and the expression e are different and $W(\langle x \rangle) \geq W(\langle e \rangle)$. A change in signedness primarily depends on the signedness of the variables in e, but also on whether or not a *signed* target variable x is able to store all possible values of the *unsigned* version of the expression e. If the target type is narrower than that of the expression, we are dealing with a truncation. In case $S(\langle e \rangle) = 1$ and $W(\langle x \rangle) > W(\langle e \rangle)$ additionally a sign-extension occurs, that is, the most significant bit of the narrower type is propagated to fill the larger width of the target variable.

```
1  short x = attacker_controlled();
2  char *buffer = malloc((unsigned short) x);
3  memcpy(buffer, src, x);
```

Figure 3: Sign-extension → buffer overflow.

Figure 3 reconsiders the example given in the introduction, illustrating a buffer overflow that is caused by a change of signedness and a sign-extension due to upcasting to a larger unsigned type (`unsigned short` to `size_t`). The example is however modified to work on all three considered platforms. An attacker controlling the variable `x` of type `short` can overflow the buffer by providing a negative number, for instance `-1`, which is converted to `0x0000ffff` in line 2 and due to a sign extension to `0xffffffff` in line 3.

source type	int	unsigned int	long	unsigned long	ssize_t	size_t/pointer	long long
dest type	4		4 (→ 8)		4 → 8		8
int	○○○	●●●	○○●	●●●	○●●	●●●	●●●
unsigned int	●●●	○○○	●●●	○○●	●●●	○●●	●●●
long	○○○	●●○	○○○	●●●	○●○	●●●	●●○
unsigned long	●●E	○○○	●●●	○○○	●●●	○●○	●●●
ssize_t	○○○	●○○	○○○	●○●	○○○	●●●	●○○
size_t/pointer	●EE	○○○	●E●	○○○	●●●	○○○	●●●
long long	○○○	○○○	○○○	○○●	○○○	○●●	○○○

Table 2: Assignments using basic types on ILP32 (left circle), LLP64 (middle circle) and LP64 (right circle): ○ denotes no problem, ● a change in signedness, possibly with sign extension (E) and ● marks a truncation.

In addition to this common type of vulnerabilities, we consider another integer flaw that has received little attention so far: For a comparison $e_1 \sim e_2$ of the two expressions e_1 and e_2 of type $\langle e_1 \rangle$ and $\langle e_2 \rangle$, an integer signedness issue occurs when the comparison is signed where it should be unsigned and an integer from the comparison is converted to an unsigned type after evaluating the expressions, or vice versa. Signedness issues of this kind depend on all properties of the compared types $\langle e_1 \rangle$ and $\langle e_2 \rangle$, that is, sign, rank, and width.

```
1  int x = attacker_controlled();
2  unsigned short BUF_SIZE = 10;
3  if(x >= BUF_SIZE)
4          return;
5  memcpy(buffer, src, x);
```

Figure 4: Signed comparison → buffer overflow.

Figure 4 shows an example of an integer signedness issue when comparing variables of different signedness resulting in a buffer overflow. In this example, the attacker-controlled variable `x` is of type `int`, and it is compared to a buffer size stored in a variable of type `unsigned short` to avoid buffer overflows. `x` bytes are subsequently copied into the buffer. Unfortunately, the comparison is signed, and therefore, if an attacker chooses a negative number for `x`, the check is by-passed, and the negative number is converted into the unsigned type `size_t` as it is passed to `memcpy` as a third argument.

3. 64-BIT MIGRATION VULNERABILITIES

Integer-related flaws have been studied in great detail in the past and several methods for analysis, detection and mitigation have been proposed [e.g., 6, 9, 30, 42, 45]. All of these approaches, however, consider defects *directly* created by the developer, such as incorrect type casts. By contrast, we focus on flaws that are introduced by migrating code to a 64-bit data model and are non-existent on the originating data model. These defects are introduced *indirectly* and are hard to spot by the developer without anticipating a later migration.

In the following, we characterize different 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 3.1) and the larger address space available on 64-bit systems (Section 3.2). For a specific data model M, we denote the width of an integer type T by $W_M(T)$ and the range by $I_M(T)$.

3.1 Effects of Integer Width Changes

All types of integers available on 32-bit platforms also exist in 64-bit data models, however, their width may differ (see Section 2.2). These changes introduce 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.

3.1.1 New Truncations

As discussed in Section 2.3, a truncation occurs when an expression is assigned to a type narrower than that of the expression itself. Table 2 provides an overview of integer issues caused by assignments, broken down by basic integer types. For each of the three prevalent data models, truncations are marked by a filled circle (●). 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 truncation, the migration of the data model additionally introduces two vulnerability patterns related to the handling of pointers.

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 $W_{\text{ILP32}}(\texttt{int}) = W_{\text{ILP32}}(\texttt{ptrdiff\_t})$, but fatal on LP64 and LLP64, where $W_M(\texttt{int}) < W_M(\texttt{ptrdiff\_t})$ for $M \in \{\text{LP64}, \text{LLP64}\}$, which means that the difference is truncated to 32 bit and thus may cause loss of information.

Figure 5 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 only 32 bit. 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.

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 3.2.1).

right operand	int	unsigned int	long	unsigned long	ssize_t	size_t/pointer	long long
left operand	4		4 (→ 8)		4 → 8		8
int	○○○	●●●	○○○	●●●	○○○	●●●	○○○
unsigned int	●●●	●●●	●●○	●●●	●○○	●●●	○○○
long	○○○	●●○	○○○	●●●	○○○	●●●	○○○
unsigned long	●●●	●●●	●●●	●●●	●●●	●●●	○○●
ssize_t	○○○	●○○	○○○	●●●	○○○	●●●	○○○
size_t/pointer	●●●	●●●	●●●	●●●	●●●	●●●	○●●
long long	○○○	○○○	○○○	○○●	○○○	○●●	○○○

Table 3: Comparisons using basic integer types on ILP32 (left circle), LLP64 (middle circle) and LP64 (right circle): ○ denotes a *signed* and ● an *unsigned* comparison.

```
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);
```

Figure 5: Example of a 64-bit migration vulnerability caused by incorrect pointer differences.

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, that is $W_{\text{ILP32}}(\texttt{int}) = W_{\text{ILP32}}(\texttt{pointer})$. In contrast, on LP64 and LLP64 this practice leads to *latent pointer truncations* [see 29] as $W_M(\texttt{int}) < W_M(\texttt{pointer})$ for $M \in \{\text{LP64}, \text{LLP64}\}$. 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) [5].

3.1.2 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. The latter case can be formalized as follows: Let M_1 be a 64-bit data model and M_2 be the 32-bit reference data model. For the assignment $x = e$, sign extensions with successive unsigned interpretation is performed in M_1, but not in M_2 if $0 = S(\langle x \rangle) \neq S(\langle e \rangle) = 1$ and $W_{M_1}(\langle x \rangle) > W_{M_1}(\langle e \rangle) = W_{M_2}(\langle x \rangle) = W_{M_2}(\langle e \rangle)$.

Table 2 indicates signedness errors in assignments as gray circles (●), where sign extensions are additionally marked (Ⓔ). 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. The example of the *zlib* vulnerability presented in the introduction illustrates this issue in a real-world scenario (see Section 5 for further details).

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. Those cases can be formally summarized as follows: Let M_1 be a 64-bit data model and M_2 be the 32-bit reference data model. Then, a comparison $a \sim b$, where (without loss of generality) $0 = S(\langle a \rangle) \neq S(\langle b \rangle) = 1$ is unsigned in M_2, but signed in M_1 if $R(\langle b \rangle) > R(\langle a \rangle)$ and $\max I_{M_2}(\langle a \rangle) \notin I_{M_2}(\langle b \rangle)$ and $\max I_{M_1}(\langle a \rangle) \in I_{M_1}(\langle b \rangle)$, that is signed type $\langle b \rangle$ can not hold the maximum value of the unsigned type $\langle a \rangle$.

Table 3 provides an overview of the signedness of comparisons for basic integer types and different data models. Unsigned comparisons are marked as filled circles (●) while signed comparisons are indicated by empty circles (○). Of particular interest are those cases where the indicators change between 32-bit and 64-bit data models. For instance, a comparison involving `long` and `unsigned int` is unsigned on both, ILP32 and LLP64, but signed on the LP64 data model.

Figure 6 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 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.

```
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);
```

Figure 6: A check to avoid buffer overflows on 32-bit systems that is ineffective on LP64 platforms.

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.

3.2 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 model in the first place, but cannot be triggered on the corresponding platforms in practice.

3.2.1 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  }
```

Figure 7: Buffer overflow resulting from an integer overflow due to larger strings on 64-bit platforms.

Figure 7 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`, that is, $W_M(\texttt{unsigned int}) < W_M(\texttt{size\_t})$, $M \in \{\text{LP64}, \text{LLP64}\}$. If the attacker-controlled value `len` is larger than `UINT_MAX` ($\texttt{len} > \max I_M(\langle \texttt{i} \rangle)$), 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 $\texttt{len} \leq \max I_{\text{ILP32}}(\langle \texttt{i} \rangle)$—in other words, the loop terminates before `i` overflows.

Vulnerabilities resulting from the large number of objects are typically tied to reference counters with a type smaller than the pointer size. We provide examples of previously unknown vulnerabilities in the *Boost C++ Libraries*, *Chromium* and the *GNU Standard C++* from this class in Section 5.

3.2.2 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 return value 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 [20, 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.

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

Figure 8: Buffer overflow caused by the common pattern of assigning the result of `strlen` to an `int`.

Figure 8 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 of length l, where $\max I_M(\langle len \rangle) < l < \max I_M(\texttt{unsigned int})$, the return value stored in `len` is mistakenly interpreted as a negative number and consequently 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.

3.2.3 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 [18, Sec. 7.19.6]. In practice, this can result in vulnerabilities when programmers directly make use of the return value to shift pointers. Figure 9 exemplarily shows a vulnerable implementation of a log function that writes messages to a global buffer of `BUF_LEN + 1` bytes in size.

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 4: 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.

```
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, "%s", str);
6
7    if(n > BUF_LEN -pos) {
8          pos = BUF_LEN;
9          return -1;
10   }
11   return (pos += n);
12 }
```

Figure 9: Stack-corruption vulnerability on 64-bit systems due to unexpected behavior of `snprintf`.

```
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;
```

Figure 10: Buffer overflow for files larger than `UINT_MAX` caused by unexpected return value of `ftell`.

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.

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, in this case, 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 (Section 4). 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 [18, 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.

Figure 10 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.

4. 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 4.1). Second, based on the observations made in the previous section we refine our search and automatically look for *programming patterns* that are characteristic for 64-bit migration flaws (Section 4.2).

4.1 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 [25]. 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 conversion that involve a loss in floating point precision [11].

Table 4 summarizes the results. We list the total count of warnings of each conversion type 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, respec-

source type	unsigned			signed				Total
destination type	size_t	unsigned long	uintptr_t	ssize_t	long	intptr_t	ptrdiff_t	*
int	10,181	4,188	0	3,294	12,114	24	6	**29,807**
unsigned int	10,989	2,857	0	91	1,339	0	0	**15,276**
int32_t	55	42	0	0	32	0	0	**129**
uint32_t	302	231	0	0	120	0	0	**653**
Total	**21,527**	**7,318**	**0**	**3,385**	**13,605**	**24**	**6**	**45,865**

Table 5: Number of width conversions in Debian stable, critical to the migration of code from ILP32 to LP64.

Code-base	P1: atol	P2: memcpy	P3: loops	P4: strlen	P5a: snprintf	P5b: ftell
Debian Jessie	21.49% (133)	7.76% (2,536)	8.47% (1,264)	13.85% (7,595)	27.55% (762)	64.74% (628)
GitHub	18.66% (25)	15.19% (2,918)	12.56% (658)	22.54% (3,572)	34.79% (502)	85.05% (182)
Average	20.98% (158)	10.51% (5,454)	9.53% (1,922)	15.80% (11,167)	30.03% (1,264)	68.41% (810)

Table 6: 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.

tively. 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 3.1.2 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 per package 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.

Finally, we look at implicit type conversion caused by the migration process from 32-bit to 64-bit platforms in more detail. Table 5 shows the absolute number of warnings for basic types that are 4 bytes wide for Debian 32 bit (ILP32), but are 8 bytes wide one Debian 64 bit (LP64), hence representing reasonable suspects for 64-bit migration vulnerabilities. In total we record more than 45,000 warnings of this kind, suggesting a huge potential for misuse. Especially, the conversion from `size_t` to `int` and vice versa appears to be problematic in practice, spawning over 21,000 warnings in core packages of Debian stable.

4.2 Patterns of 64-bit Migration Issues

Of course, not all implicit type conversions indicate a bug or even a vulnerability. We hence narrow down this vast amount of suspect locations by specifically looking for 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 3 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.

P1. *New truncations.* As an example for a truncation that exclusively happens on 64-bit systems we consider the faulty use of the standard library function `atol`. We count all occurrences of `atol` and relate these to those invocations that assign the return value to a variable of type `int` rather than `long`. In Section 5.1 we discuss a real-world vulnerability based on such a truncation.

P2. *New signedness issues.* When used in the context of memory operations, signedness issues may cause severe security flaws. For this class, we focus on unexpected sign-extensions in combination with the memory copy operation `memcpy` as discussed in Section 3.1.2. We count all invocations of `memcpy` that use a signed variable of type `int` to specify the amount of data to copy and compare these to the overall number of calls to `memcpy`.

P3. *Dormant integer overflows.* Integers may under or overflow in various situations. For this pattern we choose a rather strict scenario, in which a loop is iterating over code based on 64-bit related data. In particular, we count `for` loops that use a loop-variable of type `size_t` and relate these to the subset of loop that additionally increment or decrement a variable of type (`unsigned`) `int` in their loop body.

P4. *Dormant signedness issues.* For this class, we take up the example of the incorrect usage of the `strlen` function presented earlier. To this end, we count the calls to `strlen` that do not use a string literal as parameter and put these in relation to occurrences that assign the return value of `strlen` to (`unsigned`) `int`. Left values of other types are not considered as problematic in this example and count for the reference quantity.

P5. *Unexpected behavior of library functions.* Finally, we inspect two examples of library functions that behave differently than a) developers might expect or b) the C99 standard specification. First, we count occurrences of `snprintf` that make use of their return value and put these in relation to occurrences that use the return value but *do not* check its validity. Second, we count the calls to `ftell` in relation to the absolute usage of `ftell*` functions (`ftell`, `ftello`, `ftell64` and `_ftelli64`).

The presented patterns do not provide a complete list of 64-bit migration flaws, but should convey a feeling for the prevalence of such flaws by example. Surprisingly, these trivial patterns already point out a large number of potential issues. Table 6 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 in 30% and 70% of all cases, respectively (P5a & P5b).

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.

5. CASE STUDIES

Finally, we discuss 64-bit migration vulnerabilities from all categories described in Section 3 in practice. We build general patterns for the control-flow and data-flow of such issues, look for these in popular code bases to identify potentially vulnerable program code and manually inspect these. This effort has resulted in 6 previously unknown vulnerabilities[1] in high-quality software such as *Chromium*, the *Linux kernel* and *zlib*. Additionally, we complement our study with two vulnerabilities disclosed in the past.

In this context, we highlight the two main sources of 64-bit migration flaws: 1) changes in integer widths, and 2) the increased amount of memory available on 64-bit systems. Table 7 summarizes our findings with respect to these two categories.

Case Study		Width Change	Mem
PHP	CVE-2007-1884	×	
libarchive	CVE-2013-0211	×	
zlib	*new*	×	×
libarchive	*new*		×
Chromium	*new*		×
GNU C Library	*new*		×
Linux Kernel	*new*		×
Boost C++ Libraries	*new*		×

Table 7: Vulnerabilities discussed in this section.

5.1 New Truncations

To begin with, we briefly describe two vulnerabilities related to new truncations caused by the migration from 32 bit to 64-bit systems. In particular, we examine a vulnerability from 2007 in *PHP* as well as a vulnerability in the *Linux Kernel* discovered as part of our research.

PHP. Esser [10] describes a vulnerability that allows code execution in versions of *PHP* earlier than 4.4.5 and 5.2.1 (CVE-2007-1884). The vulnerability results from an integer truncation not present on 32-bit systems. While *PHP*'s `php_sprintf_getnumber` function, a function used by all `printf` variants for parsing format strings, returns integers of type `long`, its result is stored in variables of type `int` when processing argument numbers, width and precision. On systems using ILP32 and even LLP64 (Windows) this is not an issue as both types are of the same width. However, for LP64, `long` is eight bytes wide, introducing a truncation that can be exploited for arbitrary code execution by specifying a precision of `INT_MAX` characters.

Linux kernel. The Linux kernel comes with its own implementation of C standard library functions, e.g., for string manipulation. In contrast to corresponding implementations in the *GNU C Library*, the implementation of the *Linux Kernel* version 4.6 and before does not check for overly large inputs passed to the `snprintf` function. In particular, the function subtracts one pointer from another, yielding a value of type `ptrdiff_t`, but returns it as 32 bit wide integer. On 64-bit systems `ptrdiff_t` is 64 bit wide such that the return value is truncated for large inputs. This matches the example given in Section 3.1.1 exactly.

5.2 New Signedness Issues

To demonstrate the practicability of signedness issues on 64-bit systems, we again inspect a vulnerability discovered by us in *zlib* and one in *libarchive* reported in 2013. The first involves a sign-extension and the second, a sign-comparison issue as described in Section 3.1.2.

Sign-extension issue in zlib. This vulnerability resides in version 1.2.8 of *zlib* and is triggered by the `gzprintf` function, which is used to write a formatted string into a buffer, with a size previously specified using the library function `gzbuffer`. Unfortunately, the function is not fit to process inputs larger than `INT_MAX` bytes. While at initialization the target size is specified as `unsigned int`, the `gzvprintf` function internally casts this value to `int`. Later on this value is used as second parameter to the `vsnprintf` function which is defined as `size_t` and specifies the number of bytes to write into the buffer at most. On `LP64` and `LLP64`, `size_t` is twice as wide as `int` such that a sign extension is performed and the conversion yields a large unsigned value, allowing to overflow the buffer.

Sign-comparison issue in libarchive. Yamaguchi [39] discovered a vulnerability in the `archive_write_zip_data` function of *libarchive* version 3.1.2 and below (CVE-2013-0211). This function is used as a callback for writing zip archives and receives the destination buffer and its size as arguments. Before data is written, the implementation checks, whether the specified buffer exceeds the maximum number of bytes that may be written to the zip archive. In this context, the size is explicitly casted from `size_t` to `int64_t`, a conversion that has been unproblematic on 32 bit systems (`INT64_MAX` > `SIZE_MAX`) but changes signs on 64 bit platforms as `size_t` and `int64_t` are of the same width. Later on, the `UINT_MAX` bytes of the provided input are deflated to the output archive, irrespective of the maximally allowed number of bytes.

5.3 Dormant Integer Overflows

We proceed to give examples for integer overflows that already exist on 32-bit data models, but can only be triggered on 64-bit platforms due to the increased size of the address space. We have discovered vulnerabilities of this kind in a function of the *GNU C Library*, the shared pointer implementations of the *Boost C++ Libraries*, *Chromium*, and the *GNU Standard C++ Library*.

GNU C Library. The `wcswidth` function as specified by the "X/Open Portability Guide" [17] and implemented by version

[1]All discovered vulnerabilities have responsibly been disclosed to the vendors of the affected software projects.

2.23 of the *GNU C Library* contains an integer overflow. This function counts the number of columns needed to represent a wide-character string. This counter, however, is internally defined as a variable of type `int`, in accordance with the functions return value. As the input string might be larger than `INT_MAX` on 64-bit systems, processing the complete string overflows the counter. With a string longer than `UINT_MAX` the return value wraps around to a positive number again, such that using this value for memory allocations inevitable results in a buffer overflow.

C++ shared pointers. Shared pointers are containers for raw pointers that keep track of the ownership of these and manage the number of references. If all references are destroyed, the last instance takes care of also destroying the hosted raw pointer. Unfortunately, the implementations as distributed with version 1.60 of the *Boost C++ Libraries* (`boost::shared_ptr<T>`) as well as those of *Chromium* version 52.0 and the *GNU Standard C++ Libraries* that come with *GCC* version 6.1.0 suffer from a flaw: The reference counter is implemented as an integer of type `int`. Consequently, on 64-bit systems, an attacker may create lots of shared pointers such that the internal counter overflows until it contains a value of 1 again. The next shared pointer that is subsequently destroyed then also destroys the shared raw pointer, leaving `UINT_MAX` instances behind pointing to a freed location. This results in a use-after-free vulnerability that can be exploited by attackers for arbitrary code execution.

5.4 Dormant Signedness Issues

Finally we examine a vulnerability we discovered in *libarchive* version 3.2.0, which contains a signedness issues that, on a 32-bit system, could not be triggered as of the limited address space, but are exploitable on 64 bit.

This vulnerability in the processing of `iso9660` containers rests on checks for the maximally allowed length of Joliet identifiers. In the course of these checks, the length of the name which is stored as `size_t` is explicitly casted to `int`, very much like the example given in Section 3.2.2. It is hence possible to provide a string just long enough to change the signedness of the integer to bypass this check. Allocating as much memory, however, has only become possible with 64-bit systems. *libarchive* maintains UTF-16 and multi-byte versions of the names and therefore, allocates at least 3× more memory as theoretically needed for merely bypassing the check.

6. DISCUSSION

Ideally, flaws induced by the migration from 32-bit to 64-bit platforms are addressed by thorough code audits that specifically focus on problematic type conversions and related code patterns. However, more than 10 years after 64-bit processors have reached the mass market, we have shown that vulnerabilities resulting from the migration process still are a major issue. We thus discuss strategies to cope with this lasting issue in software security. We differentiate countermeasures based on the two root causes of 64-bit migration vulnerabilities: 1) the vast amount of memory addressable by a single process and 2) the change in width of integral integer types and resulting conversion problems.

Memory monitoring. As discussed in Section 3.2, 64-bit migration flaws are often triggered by large amounts of memory allocated by a single process. Consequently, such vulnerabilities can be coped with by tightly monitoring the usage of memory. Bernstein, for instance, even argues that the vulnerabilities in qmail discovered by Guninski [13] are not relevant at all because *"nobody gives gigabytes of memory to each qmail-smtpd process, so there is no problem with qmail's assumption that allocated array lengths fit comfortably into 32 bits"* [3]. While this does not hide the fact that qmail is vulnerable in this particular scenario, there is some truth to pointing out alternatives to limit a process' usage of resources such as `softlimit` from daemontools [4], `ulimit` [23] or `cgroup` [14].

These mechanisms are rather strict with respect to enforcing limitations and refuse to provide more memory or even kill the process (`SIGKILL`) if hard limits are reached. The use of *memory warnings*, that are raised at runtime whenever a process uses more than the granted amount of memory might be an alternative option: Instead of refusing any more memory or terminating the process, a signal (e.g., `SIGUSR1`) is sent to the process, which then decides whether or not this usage is legitimate or not. This follows the basic rationale that, although a process in principle might consume large amounts of memory (e.g., audio/video editing, image processing, scientific computing), its legitimacy depends on the task at hand and thus, is best judged by the process itself. However, it is questionable whether a process under attack still is trustworthy enough to make this decision.

Improved error reporting. As demonstrated in Section 4.1 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 1,798 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 [16], 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 analyses come at a computational cost that often does not fit the requirements of a performance-oriented compiler framework.

In conclusion the most effective strategy against 64-bit migration issues, and vulnerabilities arising thereof, is to raise awareness for this particular aspect of software security and avoid implicit conversion in the first place.

7. RELATED WORK

The discovery and prevention of integer-related vulnerabilities is a long-standing topic in computer security. In this section, we provide an overview of past research on integer-related vulnerabilities, including previous work on vulnerabilities related to 64-bit platforms, and more ad-hoc guides for safely porting code to 64-bit platforms published outside of academia.

Integer-related vulnerabilities. Many researchers have dealt with integer-related defects and vulnerabilities in the past, where efforts have been made to both identify and prevent these flaws. On the one hand, methods that operate on source code or intermediate program representations have been proposed, both based purely on static analysis [2, 24, 43], and dynamic approaches that rely on checks at runtime [6, 8, 9, 34, 35, 38, 44]. On the other hand, methods to identify these flaws in binary executables have been presented, again, both via static analysis [12, 42, 45], and in combination with dynamic approaches [7, 30, 36].

Another strain of research aims to address the root causes of integer-related vulnerabilities by designing libraries for safe integer operations, e.g., SafeInt [22] and IntSafe [15]. Moreover, dialects of the C programming language that provide additional annotations or introduce new concepts to prevent security flaws have been proposed [19, 31, 32]. While these approaches cover many different integer-related flaws, only few provide a general view on the prevalence of such flaws [e.g., 6, 9, 34, 43] and none so far, specifically address integer flaws that arise from the migration to 64-bit platforms—a gap we attempt to close with this work.

64-bit migration guides. Several informal guides for porting software to 64-bit architectures are available to date. Several of these focus entirely on correct pointer to integer conversions, the most obvious problem introduced by the new data model [37]. Moreover, large software vendors such as IBM, Oracle (Sun), and Apple have released guides for porting of code from ILP32 to LP64 that discuss generic problems, including effects on type conversions, on memory layout (e.g., the alignment of data types) and on missing function prototypes [1, 26, 33]. However, none of these guides provide a discussion of the security implications these problems have as well as discusses how exploitable vulnerabilities can emerge from the migration to 64-bit systems.

Tools for identifying 64-bit vulnerabilities. Tools to identify some vulnerabilities related to 64-bit systems have been developed. In particular, the *Viva64 static analyzer* (part of PVS-Studio) employs a set of rules for the discovery of certain 64-bit vulnerabilities [41]. Moreover, Medeiros and Correia [28] propose a tool to detect 64-bit migration vulnerabilities. Their method is based on a combination of type checking and taint tracking to pinpoint integer manipulation issues involving tainted data. The authors report that all findings in their empirical evaluation are false positives, though.

8. CONCLUSIONS

Migrating software to different platforms is a notable challenge for software security. Due to the use of different data models, assumptions about widths of integer types made for one platform do not necessarily hold true for another. In this paper, we systematically categorize and define vulnerabilities made possible by the migration process to 64-bit platforms. We show that more than 10 years after 64-bit architectures have reached the mass market, implicit conversion of types that have changed due to the different data models still are prevalent in mature and well-tested software. We find that on average, C/C++ projects from Debian stable tagged as *Required*, *Important* or *Standard* spawn 1,798 warnings concerning type conversions, 703 of which are exclusive to 64-bit systems. For example, to a large extend developers appear to unjustifiably treat the unsigned type `size_t` and (`unsigned`) `int` as equal, leading to 21,527 warnings in total, which creates a large potential for security flaws. Moreover, we look out 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.

Finally, we make use of this systematization and the experience thus gained to uncover 6 previously unknown vulnerabilities in popular software projects, such as the *Linux kernel*, *Chromium*, the *Boost C++ Libraries* and the compression libraries *libarchive* and *zlib*—all of which have emerged from the migration from 32-bit to 64-bit platforms.

Acknowledgments

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) as well as the German Research Foundation (DFG) under project DEVIL (RI 2469/1-1).

References

[1] H. S. Adiga. Porting linux applications to 64-bit systems. http://www.ibm.com/developerworks/linux/library/l-port64/index.html, 2006.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. of IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[3] D. J. Bernstein. The qmail security guarantee. https://cr.yp.to/qmail/guarantee.html, visited August 2016.

[4] D. J. Bernstein. The softlimit program. http://cr.yp.to/daemontools/softlimit.html, visited August 2016.

[5] 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.

[6] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. X. Song. RICH: Automatically protecting against integer-based vulnerabilities. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2007.

[7] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie. BRICK: A binary tool for run-time detecting and locating integer-based vulnerability. In *Proc. of International Conference on Availability, Reliability and Security*, pages 208–215, 2009.

[8] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime environment driven program safety. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, pages 385–406, 2004.

[9] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proc. of International Conference on Software Engineering(ICSE)*, pages 760–770, 2012.

[10] S. Esser. PHP printf() family 64 bit casting vulnerabilities. http://www.php-security.org/MOPB/MOPB-38-2007.html, 2007.

[11] Free Software Foundation, Inc. Warning options - using the gnu compiler collection (gcc). https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html, visited August 2016.

[12] P. Godefroid, M. Y. Levin, and D. Molnar. Active property checking. In *Proc. of ACM International Conference on Embedded Software (EMSOFT)*, pages 207–216, 2008.

[13] G. Guninski. 64 bit qmail fun. http://www.guninski.com/where_do_you_want_billg_to_go_today_4.html, 2005.

[14] T. Heo. Control group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2015.

[15] M. Howard. Safe integer arithmetic in c. http://blogs.msdn.com/b/michael_howard/archive/2006/02/02/523392.aspx, visited August 2016.

[16] IBM Corp. XL C/C++: Optimization and programming guide. Technical report, IBM Corp., 2012.

[17] IEEE and The Open Group. The open group base specifications issue 7. Technical Report IEEE Std 1003.1, IEEE and The Open Group, 2013.

[18] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[19] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 275–288, 2002.

[20] 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.

[21] T. Lauer. *Porting to Win32^{TM}: A Guide to Making Your Applications Ready for the 32-Bit Future of Windows^{TM}*. Springer, 1996.

[22] D. LeBlanc. Safeint. https://safeint.codeplex.com, visited August 2016.

[23] Linux Programmer's Manual. ulimit - get and set user limits. http://man7.org/linux/man-pages/man3/ulimit.3.html, visited August 2016.

[24] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinhard. Sound input filter generation for integer overflow errors. In *Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 439–452, 2014.

[25] M. López-Ibáñez and I. L. Taylor. The new Wconversion option. https://gcc.gnu.org/wiki/NewWconversion, visited August 2016.

[26] Mac Developer Library. Making code 64-bit clean. https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/64bitPorting/MakingCode64-BitClean/MakingCode64-BitClean.html, 2012.

[27] J. R. Mashey. The long road to 64 bits. *ACM Queue Magazine*, 4(8):24–35, 1996.

[28] I. Medeiros and M. Correia. Finding vulnerabilities in software ported from 32 to 64-bit CPUs. In *Proc. of Conference on Dependable Systems and Networks (DSN)*, 2009. (fast abstract).

[29] Microsoft Security Research and Defense Blog. Software defense: mitigating common exploitation techniques. http://blogs.technet.com/b/srd/archive/2013/12/11/software-defense-mitigating-common-exploitation-techniques.aspx, 2013.

[30] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proc. of USENIX Security Symposium*, pages 67–82, 2009.

[31] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 128–139, 2002.

[32] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[33] Oracle. Guidelines for converting to LP64. https://docs.oracle.com/cd/E18752_01/html/816-5138/convert-19.html, 2005.

[34] M. Pomonis, T. Petsios, K. Jee, M. Polychronakis, and A. D. Keromytis. IntFlow: Improving the accuracy of arithmetic error detection using information flow tracking. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 416–425, 2014.

[35] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *Proc. of International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[36] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 473–486, 2015.

[37] Software Engineering Institute, CERT Division. Converting a pointer to integer or integer to pointer. https://www.securecoding.cert.org/confluence/display/c/INT36-C.+Converting+a+pointer+to+integer+or+integer+to+pointer, 2016.

[38] H. Sun, X. Zhang, C. Su, and Q. Zeng. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 483–494, 2015.

[39] The MITRE Corporation. CVE-2013-0211. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0211, 2013.

[40] The Open Group. 64-bit and data size neutrality. http://www.unix.org/version2/whatsnew/lp64_wp.html, 2000.

[41] Viva64. Detect 64-bit portability issues. http://www.viva64.com/en/viva64-tool/, visitied February 2016.

[42] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2009.

[43] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–177, 2012.

[44] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, pages 71–86, 2010.

[45] 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 *Proc. of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 247–269, 2015.