Brave New 64-Bit World

An MWR InfoSecurity Whitepaper

2nd June 2010

MWR INFOSECURITY

MWR INFOSECURITY

Abstract

Memory requirements on server and desktop systems have risen considerably over the past few years, to the point where 32-bit architectures are not capable of addressing the required amount of memory. A variety of 64-bit CPUs and operating systems have been introduced to resolve this architecture imposed limitation and these are now being widely adopted. However, any porting of software to 64-bit compatibility can have unexpected security implications, even without any code changes in the programs, drivers or operating systems. This is particularly dangerous in situations where code has already been subject to code review and been assessed to be free from exploitable vulnerabilities in a 32-bit environment as it could immediately become vulnerable when compiled on a 64-bit system. Consequently, it is important that there is an appreciation of the security implications of the porting process and that appropriate security reviews are conducted. This whitepaper discusses the most common problems associated with code running on 64-bit systems, their impact on the security of systems and methods for preventing them.

MWR INFOSECURITY

Contents

MWR INFOSECURITY

1 Introduction

With the wide availability of x64 CPUs, many organisations are now switching to 64-bit operating systems and applications. This is driven by the increasing memory requirements of applications and servers, the decreasing cost of the new hardware and what is now wide support within applications and operating systems.

When code reviews are conducted of C/C++ applications which were developed on 32-bit systems and then ported to 64-bit, certain classes of security vulnerability are commonly identified. A number of these classes of vulnerability are discussed within this document.

It should be noted that these classes of vulnerability are not new and similar issues have been found and exploited before. However, the migration to 64-bit technology is regularly leaving organisations exposed to risk, particularly when there is a reliance on security reviews and assurance activities performed previously on a different architecture.

MWR INFOSECURITY

2 Vulnerability Concepts

2.1 Large Input

On 32-bit systems the amount of possible input to an application is naturally limited by the available address space. For example, on Microsoft Windows systems memory allocations in user-mode are usually less than 2 gigabytes in size. In reality, however, the space available for memory allocations on 32-bit systems will be much less, as space will be reserved for binaries, stacks and heaps. Nevertheless, this can still be more than 2 gigabytes when the /3GB switch is used during booting, although this is not the default setting.

However on 64-bit systems these limits are greatly increased and allocation of much larger memory blocks may be possible. This is especially true on server systems, but is increasingly common on desktop systems, where at least 2 GB of RAM is common nowadays (or at least enough virtual memory space is available for these allocations).

Whilst good practice dictates that the size of any data passed to a function is checked it is often the case that developers make assumptions about the maximum possible size of that data - and these assumptions could be based on the upper limit for a memory allocation on the platform itself. When transferred to a 64-bit system these deviations from best practice can become exploitable if an attacker can introduce large amounts of data into the application. Examples of such issues are integer overflows or integer sign vulnerabilities.

During code reviews two scenarios are commonly encountered:

2.1.1 A vulnerability can be triggered by a large amount of data

Code Example 1:

```
0: unsigned int len = strlen(input);
1: unsigned int size = len+1;
2: char *buf = malloc(size);
3: memcpy(buf, input, len);
```

In the example above there is an integer overflow vulnerability on line 1 which could result in too small an allocation occurring on line 2. This could in turn cause a heap buffer overflow when line 3 executes as the memory allocation would be smaller than the size of the data. On 32-bit systems this code would not be exploitable because of the limit imposed on the maximum size of the input data by the architecture itself. However, on 64-bit systems where up to 0xffffffff bytes of data can be introduced this could be exploitable.

2.1.2 Large allocation has to succeed to trigger the vulnerability

Code Example 2 (fictional image parsing code):

```
 0: int width = readint();
 1: int height = readint();
 2: unsigned long size= width * height;
 3: if(height > 1) {
 4:   char *buf = malloc(size);
 5:   int pos = 0;
 6:   if(buf) {
 7:     char row[BUFSIZE];
 8:     if(width < sizeof(row)) memcpy(row, input+pos, width);
 9:     ...
10:   }
11: }
```

The example code above is vulnerable to a stack-based buffer overflow on line 7 as a result of a sign issue with the "width" variable. On a 32-bit system the exploitable condition that only occurs when "width" is greater than 0x7fffffff will never be reached, as the allocation on line 4 will fail. However, on 64-bit systems this example is exploitable as larger allocations are possible and thus the vulnerable code on line 8 can be reached.

2.2 Truncation in Conversion From 'long' to 'int'

On 32-bit systems, the value types 'unsigned int', 'long' and 'size_t' can be used interchangeably; however on 64-bit systems these value types are not equivalent. In situations where these have not been used in the correct manner exploitable conditions can exist.

2.2.1 Comparison of value types and sizes

The following table shows the sizes in bits of different value types on 32-bit and 64-bit systems (assumes C code compiled with the GCC compiler).

Type	32-bit GCC	64-bit GCC
Char	1	1
Short	2	2
Int	4	4
Long	4	8
size_t	4	8
long long	8	8

The use of "int" is not an appropriate choice for variables which represent data sizes, lengths and offsets on 64-bit systems as it cannot represent all possible values. However, it is known from observations made during code reviews that "int" is the most commonly used type for these values. On 32-bit systems this is not a problem as "int" and "size_t" are equivalent; however, on 64-bit systems problems will arise

which could result in vulnerabilities being present. The following C code snippet illustrates how code can be secure on a 32-bit system but vulnerable when compiled on a 64-bit system.

Code Example 3:

```
0: void example(char *input) {
1:   char buffer[1024];
2:   unsigned int length = strlen(input);
3:   if(length<sizeof(buffer)) {
4:    strcpy(buffer, input);
5:   }
6: }
```

The “strlen” function returns a length value of type “size_t” which is then assigned to an “int” type. This assignment can lead to a truncation of the return value. For example, a return value of 0x100000010 will be truncated to the “int” value 0x10. This would result in the check being passed and a very large string being copied to the stack-based buffer, resulting in a potentially exploitable condition.

3 Proof of Concept Examples

3.1 Sendmail 8.14.4 str_union Vulnerability

MWR InfoSecurity have researched this topic further and one of the findings was that the current Sendmail implementation is vulnerable to a bug which requires very large input to be stored in memory. The "str_union" function is used in the usersmtp.c file to concatenate the values of multiple authentication responses during the extended "hello" process of an SMTP conversation. The allocation of memory for the resulting string is implemented as follows:

usersmtp.c

```
0: str_union(s1, s2, rpool)
1: ...
2: {
3:     int l1, l2, rl;
4:     ...
5:     l1 = strlen(s1);
6:       l2 = strlen(s2);
7:       rl = l1 + l2;
8:       res = (char *) sm_rpool_malloc(rpool, rl + 2);
9:       if (res == NULL)
10:      {
11:             if (l1 > l2)
12:                     return s1;
13:              return s2;
14:      }
15:      (void) sm_strlcpy(res, s1, rl);
16:      ...
```

As can be seen from the code "s1", which is the result of previous concatenations, could theoretically grow indefinitely. Although the length of each response line is limited, the number of auth response lines is not. As a consequence an attacker could make the signed integers on lines 5 or 7 wrap, resulting in the allocation made on line 8 being too small and so in turn leading to a heap buffer overflow on line 15 or later in the code.

An attacker could make a vulnerable Sendmail server connect back to a malicious SMTP server by sending an email to the domain hosting the malicious server; the vulnerability could then be triggered by sending an "EHLO" response of the following form to the target server:

```
250-local.sendmail.ORG Hello localhost [127.0.0.1], pleased to meet you
250-AUTH XXXXXXX..XXXX1
250-AUTH XXXXXXX..XXXX2
250-AUTH XXXXXXX..XXXX3
.... a few million of these lines ....
250 HELO
```

Fortunately (or unfortunately, from the perspective of an attacker) this vulnerability is not currently exploitable on Sendmail due to a memory leak when str_union is used:

usersmtp.c

```
0:  mci->mci_saslcap = str_union(mci->mci_saslcap,
                        p, mci->mci_rpool);
```

This call to the vulnerable function will leak the previous string for each call to it and thus Sendmail will run out of memory long before reaching the exploitable integer overflow.

Even though it is very unlikely that this vulnerability could be exploited on real systems, the Sendmail developers have provided a patch which can be downloaded from: -

http://www.sendmail.org/patches/auth.2

MWR INFOSECURITY

4 Recommendations

Migration Process

As the example in this whitepaper shows, the migration of software from 32-bit to 64-bit systems can introduce new vulnerabilities, or make previously unexploitable vulnerabilities exploitable. Consequently, it is recommended that the migration process should always include a code review during which the focus should be placed on security. As we have seen, the assumptions made by programmers and used in previous code reviews may not hold true.

Code Review Process

Given the fact that applications may already have been subject to security review it is important that reviewers and security consultants are aware of the specific issues that can be manifested when migrating code. A detailed discussion on the topics of memory corruption vulnerabilities or code review techniques is beyond the scope of this whitepaper; however, the following recommendations are made to provide general guidance about identifying and resolving the types of issues which could be expected to be encountered.

1. Are there any size limits on incoming data? If not, it is very likely that the code handling the incoming data is flawed or that the functions using the input afterwards will not be coded so as to handle the data in a safe manner. Reallocation operations in network applications have proven to be particularly vulnerable (as in the Sendmail example above). In many scenarios, limiting the input data to prevent excessive amounts of memory being allocated is a reasonable control to enforce.

2. Review any usage of “int” types for length, offset and size values. Any use of a 32-bit integer for these kinds of values should be investigated as it is expected that the code will be flawed in the great majority of cases. If code is found to be affected by this issue, then each instance will need to be evaluated to determine the impact. Developers may wish to review the use of "int" in their application as a whole, and use safer types such as "long" or preferably "size_t".

3. When code is first compiled for a 64-bit platform, it is important that special attention is paid to any compiler warnings, especially those concerning truncation and casting of integer types. These can often indicate bugs which might be exploitable.

MWR InfoSecurity

5 Conclusion

MWR InfoSecurity have observed that the widespread introduction of 64-bit platforms and the consequent porting of 32-bit applications can expose several types of problem. As servers and desktops are equipped with more memory previously unexploitable vulnerabilities may become exploitable.

In addition to this, the increase in bandwidth available to individuals facilitates the exploitation of these types of vulnerability without the need to use any form of compression. On a 20Mbit upstream DSL line it will only take about half an hour to send 4 gigabytes of data. Given the potential rewards, this is not an excessive amount of time to wait to gain full access to a vulnerable application.

Local application or kernel vulnerabilities which require large amounts of memory are even more likely to be exploited, as allocating and filling 4 gigabytes of memory will only take seconds on modern systems. With the growing amount of memory available in modern servers and desktops other types of attack might also become feasible, attacks such as the overflowing of 32-bit reference counters, even without any reference leaks.

MWR InfoSecurity
St. Clement House
1-3 Alencon Link
Basingstoke, RG21 7SB
Tel: +44 (0)1256 300920
Fax: +44 (0)1256 844083
mwrinfosecurity.com