Editors: Iván Arce, ivan.arce@coresecurity.com
Elias Levy, aleph1@securityfocus.com

The Rising Threat of Vulnerabilities Due to Integer Errors

Dave Ahmad
Symantec

Vulnerability research is largely driven by trends, which begin when new classes of vulnerabilities are discovered or innovative techniques for exploiting known classes of vulnerabilities are published. For example, when attackers demonstrate that they can exploit a certain type of programming error to compromise a system's security, the result is the immediate discovery of instances of that error present in software packages. In the last few years, two trends that have emerged are format-string bugs (vulnerabilities that are due to errors in the use of "printf()" functions), and heap-based memory corruption bugs (programming errors that result in data corruption in the region of memory designated for dynamic memory allocation, or the "heap"). Once techniques for exploiting these bugs were published, dozens of vulnerabilities were rapidly discovered, exploited, and fixed.

Another trend that is on the rise is finding vulnerabilities caused by *integer errors,* mistakes a programmer makes in sensitive operations involving integer data-type variables. Those flaws include integer overflow or underflow conditions, integer comparison, and integer promotion or precision errors. Bugs caused by incorrect integer use are a fact of life for developers. Prior to 2001, vulnerabilities due to these bugs had shown up once in a while on lists such as Bugtraq. As a class of programmatic flaws, they were not considered any more serious than other occasional programming mistakes. In early 2001, it became clear that this is not the case: integer errors frequently cause security vulnerabilities.

Security researcher Michal Zalewski discovered the first significant vulnerability caused by an integer error and published his findings on 8 February 2001 (www.securityfocus.com/archive/1/161444). The vulnerability affected many secure shell (SSH) versions and allowed remote attackers to obtain full administrative privileges on an affected host without any credentials (this is the same SSH vulnerability that the character Trinity exploited in the movie *The Matrix Reloaded*). Vulnerability exploitation was widespread following that vulnerability's publication due to well-written exploit code that worked consistently on a variety of systems. Since then, researchers have sought and identified several vulnerabilities due to similar programming errors (see the "Notable Vulnerabilities Caused by Integer Errors" sidebar). At this writing, the most recent significant example is the Sendmail prescan vulnerability, which can be exploited remotely to execute attacker-supplied instructions on affected hosts.

These vulnerabilities are particularly troubling for several reasons. First, programmers can easily make mistakes when writing code dealing with integer operations—even for those who genuinely understand secure programming practices. Integer manipulation errors that resulted in security vulnerabilities have been discovered in Apache, OpenSSH, and BSD kernel code—software that is generally considered well-written by experienced programmers. Second, integer-error bugs tend to be subtle. It is not as easy to spot them during manual code review as it is to find most buffer overflow and format-string vulnerabilities. Finally, there is likely a lot of vulnerable code in production use. It appears that integer error bugs are all over the place: GNU libc, Apache, Sendmail, and Snort are just some of the technologies and applications that are vulnerable to errors involving integer operations. We will doubtless see more high-profile vulnerabilities due to integer errors in the future.

Integers: A programming staple

All data in a computer system is ultimately composed of bits. A computer system's architecture defines its basic types—the number of bits used to represent a memory address and the number of bits stored in registers for instruction operands. Computer programming languages allow for problem solving through, in part, further abstraction of these simple units into more complex types. In the C programming language, the integer data type represents an element of a subset of the set of integers. The range of possible values that a variable of integer data type can represent depends on whether the integer is signed (capa-

Notable vulnerabilities caused by integer errors

Integer errors have caused some notable vulnerabilities. They are referenced by their BugTraq number.

- Apple QuickTime/Darwin Streaming Server QTSSReflector Module Integer Overflow Vulnerability (Bugtraq ID 7659); www.securityfocus.com/bid/7659. Sir Mordred discovered this and published it on 22 May 2003.
- Sendmail Address Prescan Memory Corruption Vulnerability (Bugtraq ID 7230); www.securityfocus.com/bid/7230. Michal Zalewski discovered this and published it on 29 March 2003.
- Snort TCP Packet Reassembly Integer Overflow Vulnerability (Bugtraq ID 7178); www.securityfocus.com/bid/7178. Bruce Leidl, Juan Pablo Martinez Kuhn, and Alejandro David Weil of Core Security Technologies discovered this and published it on 15 April 2003.
- OpenBSD setitimer(2) Kernel Memory Overwrite Vulnerability (Bugtraq ID 5861); www.securityfocus.com/bid/5861. Open BSD reported this on 2 October 2002.
- Apache Chunked-Encoding Memory Corruption Vulnerability (Bugtraq ID 5033); www.securityfocus.com/bid/5033. ISS X-Force discovered this and published it on 17 June 2002.
- SSH CRC-32 Compensation Attack Detector Vulnerability (Bugtraq ID 2347); www.securityfocus.com/bid/2347. Michal Zalewski discovered this and published it on 8 February 2001.
- Sendmail Debugger Arbitrary Code Execution Vulnerability (Bugtraq ID 3163); www.securityfocus.com/bid/3163. Cade Cairns of SecurityFocus discovered this and published it on 17 August 2001.

ble of representing both positive and negative numbers) or unsigned (capable of representing only absolute values) and the integer data type's width.

In implementations of C for a 32-bit Intel architecture, integers are typically machine-word length: 4 bytes or 32 bits. In addition to the basic integer type, other types of different sizes can act as integer values with different ranges. Those are `pointers`, `char`, `short`, `long`, and `long long`. The pointer type is meant for storing memory addresses; `char` is meant for storing the value of a single character; `short` is a smaller integer type, `long` stores a longer data type (on modern Intel systems, it is equivalent to `int`); and `long long` is an integer value that is twice the size of `long`. On the 32-bit Intel platform, variables of type `pointer` are 4 bytes wide, `char` 1 byte, `short` 2 bytes, `long` 4 bytes, and `long long` 8 bytes. As a signed integer type, a variable can represent a value in an integer range that includes both positive and negative numbers with one bit representing the sign. As an unsigned integer type, a variable can represent a value in a range of absolute values using the data type's entire bit-space (for example, 0x00000000 to 0xFFFFFFFF).

Basic integers are commonly used data types. They are typically used for integer data, integer arithmetic, to store memory addresses, act as array indexes or counters, and to represent flags (as bit-fields). They are a staple of computer programming. Their uses frequently put them in security-sensitive operations, such as memory copies, buffer-size calculation, and validation checks. A handling or manipulation error of integer values can break an entire system.

Integer errors typically fall into one of three categories:

- Integer overflow and underflow conditions
- Integer comparisons
- Precision and promotion errors

This article will describe these three vulnerabilities with examples to demonstrate them.

Integer overflow and underflow

As mentioned earlier, integer variables can represent only values in a predefined range. When the unsigned integer evaluation result exceeds that range, it is reduced modulo the total number of possible values. This is known as an unsigned integer underflow or overflow condition (informally, the choice of term can depend on whether the original value was subtracted, added, or multiplied out of range).

This so-called wrap-around property can cause problems when programmers developing security-sensitive code do not anticipate it. For example, unsigned integer overflows or underflows often lead to exploitable conditions when developers use the integer in question in their calculations of how much buffer space to allocate.

The following example code contains an unsigned integer overflow condition. This small program's goal is to allocate space for an array of structures based on a user-supplied argument. It then fills the structures with data and performs whatever other operations are necessary on that data.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 struct thing {
  char  buf[100];
  int i;
 };
 unsigned int num; /* Stores number of
records */
```

```
 char *ptr = NULL;
 struct thing *records = NULL;
 unsigned int i = 0;
 if (argc < 2)
 {
  fprintf(stderr,"Number of records
required.\n");
 exit(0);
 }
 num = strtoul(argv[1],&ptr, 0); /* Read
number of records from command-line */
 if (*ptr != '\0')
 {
 fprintf(stderr,"invalid value:
    %c\n",*ptr);
 exit(0);
 }
 printf("Memory allocated: %u\n",(num *
sizeof(struct thing)));
   if ((records = malloc(num *
sizeof(struct thing))) == NULL)
 {
 fprintf(stderr,"Error allocating stor-
age for %u records\n",num);
 exit(0);
 }
 for(i = 0;i<num;i++)
 {
 /* Read data from the user into the
 structures and perform other opera-
 tions (perhaps using malloc(), free()
 and some libc functions */
 }
 free(records);
}
```

This program looks fairly solid: it checks to ensure that a valid numeric string is supplied and ensures that the attempt to allocate memory is successful. It then only writes as much data to the buffer as has been allocated—or does it?

Not always, unfortunately. To calculate the amount of memory it must allocate, the program multiplies the number of records required (a value the user supplies) by the structure's size. What the programmer of this code has not anticipated is the possibility that the product of these two values will exceed the maximum possible unsigned integer value. If this is the case, the product will be reduced modulo the number of possible values—the high-order bits that cannot be represented given the data type's width are not stored. This reduced product, less than the amount of space actually required, will be passed to `malloc()`, the C library function used to request dynamically allocated memory, as the amount argument. This will result in a smaller buffer being allocated (on Intel 32-bit systems, supplying 0x40000001 will result in allocation of only 104 bytes—or enough space for one record).

Having not detected this error, the program continues to execute. It could then write beyond the buffer over neighboring memory in the heap. Depending on what else is allocated and what other operations are performed during the life of the program, the condition is likely exploitable as a typical heap-based overflow.

Integer comparisons

Another type of integer error can occur when a program evaluates comparisons between integers with different or even ambiguous types. Subtle errors in such expressions can sometimes result in undesirable behavior.

Signed integer comparisons

An error can occur in the comparison of integer variables defined as signed. This is especially dangerous when the expression is an important operation such as a security check. A classic example of such a situation is when a program attempts to verify that a user-supplied value is less than a maximum value and both variables are declared as signed integer types. This type of operation is frequently used as a security check to, for example, prevent buffer overflows. If not prevented, an attacker could supply a value that is high enough that the CPU interprets it as negative, successfully passing such a check. Here's an example:

```
#include <stdio.h>
#define MAX 500
int main(int argc, char *argv[])
{
 int i;
 char buf[MAX];
 int max = MAX;
 if (argc != 2)
 {
 fprintf(stderr,"Argument required.\n");
 exit(0);
 }
 i = atoi(argv[1]);
 if (i < max) /* Security check! */
 {
 printf("Security check passed.\n");
 buf[i] = '\0';
 }
}
```

In this scenario, a program must use an untrusted value as an index into an array for the purpose of writing a value. This operation is very sensitive—the programmer does not want any memory beyond the bounds of the array corrupted with the zero byte—and even one byte can have se-

rious consequences that attackers can sometimes exploit. (For attacks on the stack, see "The Poison NUL Byte," Olaf Kirch's Bugtraq post [www.securityfocus.com/archive/1/10884] and klog's "Frame Pointer Overwriting" [www.phrack.org/ show.php?p=55&a=8].) Security researchers have also demonstrated that attackers can exploit off-by-one errors in the heap (www.securityfocus.com/archive/82/321607/2003-05-09/2003-05-15/1). To prevent this from happening, the code contains an attempt to perform a security check: by comparing the integer variable *i* to the variable max. If the value of the user-supplied variable *i* is large enough, the CPU interprets it as a negative number and the security check passes, in error. To do this, the attacker must supply a value such as 0x80000000 (in hex) or higher, which will result in a signed comparison between a positive max and – 2147483648 (for 0x80000000). Once the check has passed, a write will occur at a negative offset from the array, outside of its bounds. Depending on the circumstances, this might be an exploitable condition.

Constants and defines

More problems can arise when comparisons are evaluated against *defines,* preprocessor definitions. Defines are constants that are embedded in source code using token values known as preprocessor symbols. During the compilation process, the C preprocessor will replace all instances of these tokens in source code with their constant values, defined in a single place by the programmer beforehand. When programmers use define constants in expressions, the way the compiler interprets the constants depends on other variables in the same expression. A potentially bad situation could occur if the programmer assumes that because a definition is negative, any expression involving it will be evaluated as a signed integer operation. Unfortunately, this is not necessarily the case.

Consider the following code:

```
#include <stdio.h>
#define ERRORCODE -1
int main(int argc, char *argv[])
{
 unsigned int i;
 char *ptr = NULL;
 char *somepointer = NULL;
 if (argc != 2)
 {
 fprintf(stderr,"Argument required.\n");
 exit(0);
 }
 i = strtoul(argv[1],&ptr,0);
 if (*ptr != '\0')
 {
 fprintf(stderr,"Invalid character:
    %c\n",*ptr);
 exit(0);
 }
 /* Do some things.. */
 if (i > ERRORCODE)
 {
 /* Success! */
 exit(0);
 }
 else
 {
 fprintf(stderr,"There was an error,
 cleaning up last buffer.\n");
 free(somepointer);
 exit(0);
 }

}
```

This simple program accepts an unsigned integer parameter, and performs some operations followed by a check to ensure that *i* is greater than `ERRORCODE`, defined as –1. The programmer has obviously forgotten that *i* is declared as an unsigned integer or has assumed that the comparison will be evaluated as signed. To the compiler, this is not a comparison between *i* and –1: it is a comparison between *i* and the largest possible integer value. Therefore, *i* will always be less than or equal to `ERRORCODE`.

In this example, an error condition will always be detected. If `somepointer` had already been freed because of a successful operation, the `free()`—the C library function to release allocated memory—to clean up the last buffer might be a double free, resulting in a potentially exploitable condition.

This same potential for error is present when using numerical constants in source code, which is what define symbols such as `ERRORCODE` become after preprocessing.

Precision and promotion

Errors can also occur when a programmer defines integer variables to be the value of other integer variables that were defined as different-sized types. The programmer's failure to acknowledge adverse results of performing such operations can sometimes result in a security vulnerability.

Promotion of smaller variables

Not all integers are the same. As noted earlier, the C programming language supports `pointers`, `char`, `int`, `long`, and `long long` as basic integer types. During development, situations in which a smaller-width type variable's value must be stored in a larger-width type's variable (and vice versa) are not uncommon. Assigning the value of a variable of one type to another of a different type in the source code can cause problems if the programmer is not

fully aware of how the compiler treats such expressions.

When an expression involves integer variables of different sizes, the variable with less width is first converted to its equivalent value as a variable of greater width. For example, when an integer variable is assigned the value of a `char` variable, the `char` variable is first converted to its numeric value as an integer. This does not cause a problem when the types are unsigned because larger types with the same bit representation can represent all values of the smaller type.

It's a different story when the types are signed. For example, on a 32-bit Intel system, the representation of –1 as type signed `char` is 0xFF; the representation of –1 as a signed integer is 0xFFFFFFFF, not 0x000000FF. When a statement assigning –1 as a signed `char` variable to a signed integer value, the value assigned will be –1 as a signed integer.

As Michal Zalewski discovered, this is precisely what caused the Sendmail prescan vulnerability. The flaw existed in a routine for parsing email addresses in a loop that copied externally supplied data into a memory buffer. The loop iterated once for each character, with a security check in place to ensure that bytes were not written beyond the buffer's end. During the loop, a signed integer type was assigned the value of a signed `char` (the current character in the externally supplied source string) without explicit casting. The application would skip the security check if the integer value were equal to a constant –1, defined in the source code using the preprocessor symbol `NOCHAR`. Presumably, the programmer assumed that a byte value could never cause the integer to equal (`int`)–1. Unfortunately the assignment of (`char`)–1 to an integer type resulted in the value (int)–1 (or, `NOCHAR`) being stored. This could allow in an attacker to construct a string that could evade the security check, causing data to be written beyond the buffer's bounds.

Loss of precision

Another error that can sometimes result in an exploitable condition is when a variable defined as a smaller-size type is assigned the value of a variable defined as a type with a larger size. In these situations, the high-order bits that cannot be stored in the variable of smaller type are discarded. The stored value is reduced modulo the number of possible values; it is effectively overflowed. While similar to integer overflow or underflow errors, the situation is unique enough to warrant mention on its own.

The following code example demonstrates a possible scenario:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
 unsigned int i;
 unsigned int j;
 unsigned short size;
 char *ptr = NULL;
 char *arg = NULL;
 char *pos = NULL;
 if (argc != 3)
 {
fprintf(stderr,"Argument required.\n");
 exit(0);
 }
 arg = argv[2];
 i = strtoul(argv[1],&ptr,0);
 if (*ptr != '\0')
 {
 fprintf(stderr,"Invalid character:
     %c\n",*ptr);
 exit(0);
 }
 size = i;
 /* Imagine some code here allocating
other buffers, etc */
 if ((ptr = malloc(size+1)) == NULL)
 {
 fprintf(stderr,"Error allocating
    memory!\n");
 exit(0);
 }
 pos = ptr;
 for(j=0;j<i;j++)
 {
 *pos++ = *arg++;
 }
 ptr[i] = '\0';
 /* Do some stuff, including calling
 free, libc functions, etc */

}
```

Errors can also occur when a programmer defines integer variables to be the value of other integer variables that were defined as different-sized types.

This program accepts a length parameter and string as its arguments. The size parameter, used to store the required memory, was declared by the programmer of this code as a `short` integer. The program assigns the

Further reading

- "Reviewing Code for Integer Manipulation Errors," M. Howard, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp
- "Big Loop Integer Protection (Phrack 60)," O. Horowitz, www.phrack.org/show.php?p=60&a=9
- "Basic Integer Overflows (Phrack 60)"; blexim, www.phrack.org/show.php?p=60&a=10

value of the user-supplied length parameter, defined as unsigned integer *i*, to the size variable. Because the type of the size variable is less wide, the assigned value is reduced modulo the number of possible values of a `short` integer type. This reduced value is then passed to `malloc()` as the amount argument—less than is actually required. What follows is a standard heap overflow as bytes from the string are copied beyond the allocated buffer's boundaries.

Where to go from here?

It is not difficult to see why these vulnerabilities are a serious risk. The programming errors responsible for them are subtle—incredibly easy for even experienced programmers to make. Finding them is also difficult. These vulnerabilities are by no means exclusive to C; in fact, because C++ is a superset of C, it is just as vulnerable. Managed languages such as Java and .NET are not immune either. Integer errors in the implementation of native methods might be exposed through their interfaces.

Besides being aware during development of all variables' definitions and understanding how the compiler will interpret expressions, programmers can take some measures to mitigate the risk:

- Enable full warnings when compiling, and then heed the warnings.
- Explicitly cast variables used in expressions when the variables are different types and cast preprocessor definitions to the appropriate type when they're used in source code.
- Identify integer values that are supplied as input from external sources and follow their path in the code to find potential vulnerabilities.
- Identify iterative operations that depend on external sources—for example, a loop for reading input from remote clients.
- Examine all length and other security checks carefully.
- Use unsigned integer types if negative numbers are not necessary under any circumstances.
- Be mindful of the difference in size among types when porting or cross-compiling, and check `typedef` definitions to avoid surprises (such as `size_t`, the typedef data type meant for storing the size of data objects, defined as a signed type).
- Carefully note function prototypes and the types of their return values.
- During testing and debugging, supply extremely large/negative integer values as input parameters and observe the response of the system being tested. Although imperfect, it might uncover some of the more obvious instances of this bug.

There's no doubt that there will be significant new vulnerabilities discovered due to integer errors in any of the thousands applications in use. Unfortunately, that code has already been written. Following these guidelines and, most importantly, maintaining awareness of the risks during development will help prevent the introduction of these flaws in the future. □

David Ahmad *is manager of threat analysis at Symantec. His research interests include source code analysis, binary code analysis, and secure programming methodology. He is also moderator of the Bugtraq security mailing list, a popular security forum with more than 50,000 subscribers. Contact him at David_Ahmad@Symantec.com.*