SECURITY

Programming Language Format String Vulnerabilities

By Hal Burch and Robert C. Seacord, February 02, 2007

Are C/C++ the only languages with security vulnerabilities? What about Perl, PHP, Java, Python, and Ruby?

Hal is a Vulnerability Research at CERT. He can be contacted at www.hburch.com.
Robert C. Seacord is Senior Vulnerability Analyst for CERT/CC. He can be reached at rcs@cert.org.

Although not as well known as other vulnerability types such as buffer overflows, format string vulnerabilities have been known to exist in C and C++ programs since at least 1999, when a format string vulnerability was found in AnswerBook2 (cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-1417). Formatted output became a major focus of the security community in June 2000, when a format string vulnerability was discovered in the Washington University ftpd (WU-FTPD) software package (www.kb.cert.org/vuls/id/29823).

But format string vulnerabilities are not limited to programs written in C and C++. Other languages that include format strings include Perl, PHP, Java, Python, and Ruby. While these languages are relatively immune from buffer overflows because they maintain dynamic arrays and strings for programmers, programs written in them may still contain format string vulnerabilities.

Format string vulnerabilities result from including data from an untrusted source, such as a user, in a format string. Format strings are used by input and output routines to specify a conversion between a character string and a set of data values. The following example shows how the C function *printf()* accepts a format string and a set of values:

```
1  printf ("%s Pop: %11d\n",
2      country, pop);
```

and produces a string:

```
1  United States Pop:    295734134
```

In the format string, the % begins a conversion specification. This is followed by a set of formatting parameters and the data type. The *%s* conversion specifier instructs *printf()* to output a string value (the value passed as an argument). The *%11d* conversion specifier instructs *printf()* to output a decimal value (the "d") in an 11-character field. Format strings can be much more complicated, including flags, precisions, length modifiers, and even variable widths specified in parameters.

Directly including user input in a format string lets an attacker inject format specifications into the format string. This is particularly problematic in programming languages that support the relatively unknown *%n* specification. This unusual specification causes the number of characters successfully written so far to be stored in the integer whose address is given as the argument. If attackers can write data values to memory, they can often leverage that to gain control of the system. Even if the language does not support *%n*, an attacker may cause the format string to include more specifications than parameters. Depending on what stack protection exists in the language, an attacker may be able to access private data, avoid logging, or crash the program. (Writing exploits for a format string vulnerability is beyond the scope of this work. For a more detailed explanation, see Robert Seacord's *Secure Programming in C and C++*; Addison-Wesley, 2005.)

Accessed 01/21/26 from:
https://web.archive.org/web/20140815145809/http://www.drdobbs.com/security/programminglanguage-format-string-vulne/197002914

Format string vulnerabilities often result from a programmer being unaware that a particular routine takes a format string. For example, you can write:

```
1  snprintf(str, sizeof(str),      "Wrong password for email %s",
2          email);
3  syslog(LOG_WARNING, str);
```

Unfortunately, the *syslog()* routine uses its second parameter as a format string. As a result, if an attacker inputs an e-mail of "webmaster%s%s%s%s@example.com", *syslog()* looks for parameters to interpret the *%s* conversion specifiers in the format, most likely resulting in the program crashing. A more advanced attack may use *%n* to gain control of the system.

Another common source of format string vulnerabilities is when you need to write an error to more than one location. For simplicity, you may construct the string using *snprintf()* and then use one routine to print the message to a log and another routine to output the message to the end user in some way, such as in a message box. If either routine allows for format strings, you must be careful to include the format specification in the call:

```
1  fprintf(log, "%s", logmessage);
```

instead of neglecting it as in the following call:

```
1  fprintf(log, logmessage);
```

The first invocation is the correct one, avoiding a format string vulnerability by specifying that a string (*%s*) should be outputted and then providing that string. Because the second is shorter and may correspond to how you are thinking about the desired behavior, you may write the statement in this fashion without considering the consequences.

In this article, I explore the potential consequences of format string vulnerabilities in Perl, PHP, Java, Python, and Ruby programs.

Accessed 01/21/26 from:
https://web.archive.org/web/20140815145809/http://www.drdobbs.com/security/programminglanguage-format-string-vulne/197002914

Are C/C++ the only languages with security vulnerabilities? What about Perl, PHP, Java, Python, and Ruby?

Perl

The Perl programming language is commonly used for web applications, among others. Perl provides stack checking and other security features that make many types of vulnerabilities that are common to C programs simply not possible. Moreover, Perl provides a taint mode that marks data from untrusted sources as tainted. Some operations, such as running system commands, are not allowed on data marked as tainted. While this prevents compromise, the Perl interpreter's behavior in such cases is to exit with an error message. These protections do not, however, eliminate the possibility or risk of a format string vulnerability in Perl.

In April 2005, Stefan Schmidt reported a problem in postgrey that was a result of a format string vulnerability (cve.mitre.org/ cgi-bin/cvename.cgi?name=CVE-2005-1127). Postgrey is an anti-SPAM program that works with Postfix to implement greylisting. Stefan Schmidt was having problems with postgrey crashing on e-mail from similar e-mail addresses. These e-mail addresses were of the form *foo_bar%nowhere.com.xy@[622.622.615.619]*. The problem with this e-mail address is that, if a Perl program uses it in a format string, Perl interprets the *%n* in the format string and writes the number of characters written thus far to the memory location specified on the stack. In the case of postgrey, no other value was placed on the stack, causing the crashes.

The Perl interpreter partially protects the stack in *sprintf()* by using a special item that is marked as unwritable. The interpreter can detect when a Perl program specifies *%n* but does not pass an appropriate parameter. The result is that the program exits with a message such as "Modification of a read-only value attempted at foo.pl line 345." This causes the program to crash but does not allow the vulnerability to be exploited by attackers to execute arbitrary code by altering the stack.

Another, more serious consequence of format string vulnerabilities in Perl is illustrated by the following code example. It comes from a paper on Perl format string vulnerabilities by Steve Christey of MITRE (www.securityfocus.com/archive/1/418460/30/0/threaded), a draft of which was written in 2002.

```
1  $a = "A";
2  printf ("Before: $a\n");
3  printf ("$ARGV[0]", $a);
4  printf ("After: $a\n");
```

If this script is called *vulcode*, then *./vulcode %n* prints:

```
1  Before: A
2  After: 0
```

In fact, users could set *$a* to be any nonnegative integer value. The consequence of this flaw depends on how *$a* is later used. It may not be a vulnerability at all or it may allow attackers to subvert the program.

PHP

Format string vulnerabilities in PHP are difficult to directly exploit. PHP does not support *%n*. If the number of specifications exceeds the number of parameters, the script outputs a message of the form:

```
1  Warning: sprintf():
2      Too few arguments in
3          foo.php on line 4
```

PHP does not halt executions due to this error message, so the script continues to execute. However, the string constructed by the *sprintf()* call is blank, which may result in blank log messages or other problems, such as an attacker being able to eliminate a message via cross-site scripting.

Accessed 01/21/26 from:
https://web.archive.org/web/20140815145809/http://www.drdobbs.com/security/programminglanguage-format-string-vulne/197002914

Are C/C++ the only languages with security vulnerabilities? What about Perl, PHP, Java, Python, and Ruby?

Java

Support was added for *printf*-style format strings in Java 1.5. Java programs that use these routines may contain format string vulnerabilities. A malformed format string or insufficient arguments passed to these routines results in an exception being thrown. If the exception is not properly handled, attackers may be able to leverage the exception into a denial-of-service attack. If the exception occurs during logging, attackers may be able to prevent their activities from being logged.

Python

The Python language does not contain a *sprintf()* function but does contain the % (format) command. This command has two forms. In the first form, it acts much as *sprintf()*, taking a format string and a list of parameters. In the second form, it takes a format string and a dictionary.

Python checks the parameter list to ensure the number of parameters is equal to the number the format string specifies. In the case of a mismatch, Python generates an exception. Consequently, a format string vulnerability in a Python program results in an error message and the Python program terminating unless an error handler deals with the resulting exception. As a result, a format string vulnerability in Python may let attackers launch denial-of-service attacks or circumvent logging facilities (if the Python program crashes before logging the attack). Python does not support *%n*, so attackers cannot use format string vulnerabilities to alter variable values.

In a program using the second form, a format string vulnerability in a Python program may let attackers view entries in the dictionary that they would not otherwise be able to view. The impact of such a vulnerability depends greatly on the type of data stored in the dictionary.

Consider the following Python program:

```
1  userdata = {"user" : "jdoe",
2      "password" : "secret" }
3  passwd  = raw_input("Password: ")
4
5  if (passwd != userdata["password"]):
6      print ("Password \"" + passwd
7        + "\" is wrong for user
8            %(user)s") % userdata
9  else:
10         print "Welcome!"
```

Usually, if someone enters an incorrect password, they get a message like this:

```
1  Password "green"
2      is wrong for user jdoe
```

If attackers enter a password of *%(password)s*, the program outputs the correct password instead of the password entered:

```
1  Password "secret"
2      is wrong for user jdoe
```

By attacking the format string vulnerability, attackers can trick the program into displaying parts of the dictionary the attacker should not have access to. In this example, the attacker can discover the password.

In addition to gaining access to private data, a malicious user can cause a *KeyError* exception by entering a key without a value. In the previous example, entering a password of *%(homedir)s* would result in a *KeyError* exception. Depending on exception handling and how the resulting string was to be used, this may let attackers launch denial-of-service attacks or circumvent logging facilities.

Are C/C++ the only languages with security vulnerabilities? What about Perl, PHP, Java, Python, and Ruby?

Ruby

Similar to Python, format string vulnerabilities in programs written in Ruby can allow an attacker to terminate the program prematurely. If Ruby encounters a format specification it does not understand, such as *%z*, or if the format string contains more specifications than parameters passed to *sprintf()*, Ruby terminates the program with an error message such as "*in 'sprintf': too few arguments. (ArgumentError)*" or "*in 'sprintf': malformed format string - %z (ArgumentError)*". This can let attackers launch denial-of-service attacks or circumvent logging facilities. Ruby does not support *%n*, so an attacker cannot use format string vulnerabilities to alter variable values.

Conclusion

Format string vulnerabilities are a lesser known type of vulnerability that you should be aware of. C and C++'s support for *%n*, combined with its lack of stack protection, makes format string vulnerabilities in C and C++ programs particularly exploitable. However, format string vulnerabilities can exist in programs written in other programming languages such as Perl, PHP, Java, Python, and Ruby. Although the consequences of such vulnerabilities may not generally be as high as format string vulnerabilities in C and C++ programs, a resourceful attacker may be able to leverage the vulnerability to launch a denial-of-service attack, discover privileged information, alter variable values, or circumvent logging facilities.

Another risk of including user data in format strings is that a vulnerability in the format string parsing code in the language interpreter or a component library may be exploitable. In December 2005, Jack Louis of Dyad Security discovered a vulnerability in Perl's format string parsing routine (cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-1417). Exploiting this vulnerability required specifying a length exceeding 2,147,483,647, which is unlikely under normal conditions. On the other hand, attackers could easily use a format string vulnerability in Perl programs to specify such a length. Perl programs are not immune to format string vulnerabilities, but the vulnerability in the Perl interpreter increases the potential impact when they do occur. Louis's was not the first discovery of such a vulnerability: In 2000, the PHP interpreter was found to contain a format string vulnerability in its logging facility (cve.mitre.org/cgi-bin/cvename.cgi?name= CVE-2000-0967). Thus, even when a programming language contains protections for format string vulnerabilities, there is still risk in including user input in format strings.

It is important to realize that format string vulnerabilities can have serious security consequences. Avoiding format string vulnerabilities is two-fold:

- Be aware of which routines accept format strings and never include user data in format strings passed to those routines.
- If you want to format the string before outputting it, always use the *%s* specification and pass the string as an argument.

Acknowledgment

Thanks to Pamela Curtis for reviewing and editing this article.