The Heartbleed bug got a lot of press lately, but most of it addresses the breach and what information could be lost as well as what problems that loss would cost. Few reports have covered the actual problem and its solution.

**Related Articles**

* [Java 8 Arrives](http://electronicdesign.com/blog/java-8-arrives)
* [ARMing Ada](http://electronicdesign.com/blog/arming-ada)
* [C++11 and Ada 2012 - renaissance of native languages?](http://electronicdesign.com/embedded/c11-and-ada-2012-renaissance-native-languages)
* [Ada 2012: The Joy of Contracts](http://electronicdesign.com/dev-tools/ada-2012-joy-contracts)
* [Ada Offers Advantages Over C And C++](http://electronicdesign.com/embedded/ada-offers-advantages-over-c-and-c)

The Heartbleed bug occurs in the [OpenSSL](https://www.openssl.org/) open-source code. In particular, the fix can be found in the SSL support file d1\_both.c. The memory safety problem occurred because the system uses a mix of C strings and Pascal-style buffer-size strings. In essence, it is a buffer underrun issue versus the more common buffer overrun bug.

The packets used with the SSL protocol employ sized strings. The code to process the packets was implemented in C. The trick was to request a large amount of data, up to 64 kbytes, while having a string that was significantly smaller placed in the buffer to be sent back to the initiator. The bug is that in this case the contents of the buffer can include information from previous communication that could include details like user names and passwords.

|  |  |
| --- | --- |
| http://insidepenton.com/electronic_design/adobe-pdf-logo-tiny.png | [Download this article in .PDF format](http://electronicdesign.com/datasheet/what-heartbleed-should-teach-embedded-programmers-pdf-download) This file type includes high resolution graphics and schematics when applicable. |

This means a single attack may return useless junk, but attacks can be performed repeatedly since they are processed as valid requests. There isn’t even a hint of a problem. An attacker eventually would get useful information, and additional programming would allow the responses to be scanned automatically.

The patch to fix the bug includes a number of changes, but the critical one is shown here. It essentially ignores improperly formed requests like the ones that could have occurred before:

*/\* Read type and payload length first \*/*  
  
if (1 + 2 + 16 > s->s3->rrec.length)  
  
    return 0; */\* silently discard \*/*  
  
hbtype = \*p++;  
  
n2s(p, payload);  
  
if (1 + 2 + payload + 16 > s->s3->rrec.length)  
  
    return 0; */\* silently discard per RFC 6520 sec. 4 \*/*

The rrec.length is the record length that should have been used. The second test checks the payload size, which is where an attacker would have lied about the buffer size required to retrieve more information.

What is interesting is that the response size could have extra padding bytes at the end of the buffer. The code fills these bytes with random data. Unfortunately, if the response is smaller than the data size, then the data between the actual response and the padding would not be cleared.

This fix eliminates the flaw, but it is not the only way to solve the problem. What is not shown is the malloc used to allocate the buffer. C malloc does not clear the contents of the buffer before handing it off to the application. This is faster but not secure. Some SSL implementations utilize secure memory allocation schemes that zero a buffer before it is used. Another alternative would be using string copy operations that would clear the trailing part of a buffer.

This type of problem is not new. It does highlight the issue of security and data leaks. The program does not crash and it otherwise operates properly. Detecting the bug is not easy from a programming perspective although it was easy to fix. Static analysis tools and some programming languages like Ada and Java can address similar issues but it still is up to the developer to good, security related design practices.

**New Priorities**

Security and safety are finally in the forefront for programmers. Justifying tools or languages to help address these issues as well as reduce bugs in general is now more practical for many applications. Programmers might want to consider Java 8 *(see “*[*Java 8 Arrives*](http://electronicdesign.com/blog/java-8-arrives)*” at electronicdesign.com)* or Ada 2012 *(see “*[*Ada 2012: The Joy of Contracts*](http://electronicdesign.com/dev-tools/ada-2012-joy-contracts)*” at electronicdesign.com)*.

Static analysis tools may be a better alternative for C/C++ programmers if switching languages is not an alternative. Standards like MISRA C *(see “*[*New Version Of MISRA C: Why Should You Care?*](http://electronicdesign.com/dev-tools/new-version-misra-c-why-should-you-care)*” at electronicdesign.com)* can help prevent a significant effect on the number of bugs. Even using secure memory allocation libraries can help.

Bugs will crop up regardless of how good the programmers or their tools are. The issue should be how to minimize the number of bugs and to how to limit bad effects when they do occur. The choice of language, frameworks, runtime, and tools can help. So can programmer training. It is difficult to prevent problems if one does not know what those problems are or how to avoid them.