Published in final edited form as:

IT Prof. 2016 ; 18(6): 58–61. doi:10.1109/MITP.2016.117.

Defeating Buffer Overflow:

A Trivial but Dangerous Bug

Paul E. Black and **Irena Bojanova**

US National Institute of Standards and Technology

Abstract

The C programming language was invented more than 40 years ago. It is infamous for buffer overflows. We have learned a lot about computer science, language design, and software engineering since then. Because it is unlikely that we will stop using C any time soon, we present some ways to deal with BOF. Many of these techniques are also useful for other programing languages and other classes of vulnerabilities.

Definition and Description

The term “buffer” comes from decades ago when I/O operations were slow. Memory was set aside to hold a chunk of output data going to a device—such as a printer or a 1,200 bit/s modem—or input data being received from a keyboard or a punch card reader. When the buffer access was finished, the computer was interrupted to set up another I/O operation. The term has come to mean a chunk of contiguous memory whose values constitute a larger whole. For instance, a string is often stored as characters kept in a contiguous set of memory locations. We use the C language standard term “array,” but retain the common, although less precise term “buffer overflow.”

An array is a semistructured group of elements of the same type. The elements are accessed by integer indexes. In C, arrays are zero-based—that is, the first element has index 0. Other languages are one-based or allow the user to define the first index. In C, valid indexes range from zero to the total number of elements, minus one. Because C allows a reference (pointer) into an array, an indexed access with a negative index might be valid, too.

The Bugs Framework (BF) defines the Buffer Overflow (BOF) class as follows: “The software accesses through an array a memory location that is outside the boundaries of that array.”[1] In other words, the program uses an array reference to read from or write to a memory location that is before the beginning or after the end of the array. The BF provides information on the causes, attributes, and consequences of other bug classes, such as injection (INJ), information exposure (IEX), and control of interaction frequency (CIF).

Figure 1 shows that there are only two proximate causes of BOF: *data exceeds array* (that is, the amount of data exceeds the size of the array), or there is a *wrong index* or *pointer out of range*. These might be a result of other causes, too. *Data exceeds array* has two specific cases. In the first case, the programmer allocated the *array too small*, as in CVE-2015-0235–Ghost (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0235). The code computes the size of the needed array but leaves out one factor, which makes the array four

bytes short. In the second case, *too much data* was accessed, as in CVE-2014-0160–Heartbleed (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160). Instead of finding the length of the reply string that is already stored in an array, the code uses a number from an input. So, bad input can cause the code to read far too much data. The chain of causes for Heartbleed are *input not checked properly*, which leads to *too much data* read —specifically, a huge number of bytes are read from the heap.

Buffer overflow causes failures because data is read or written in ways that are entirely foreign to what the programmer plans. Memory contains information, such as the address of the next instruction to execute after returning from a function, calling parameters, variables used in the function, data structures, and permission flags set by the operating system. Writing outside an array could change any of these. In the worst-case scenario, adversaries could cause the program to gain extra permission or make the program execute arbitrary code. Reading beyond array boundaries could retrieve sensitive data, such as old passwords, that are left in memory after they are processed.

Detecting Buffer Overflows

Buffer overflow can be detected through two general approaches: external and internal. Internal mechanisms are those that are built into a program and operate during execution. External or static mechanisms do not access the state of executing programs. The first external mechanism is observing a program's behavior.

Almost any failure could be the result of many kinds of bugs. However, some failures have characteristics strongly suggesting BOF as the software weakness:

- Is far more information produced than expected? This suggests a read BOF. Heartbleed might have been discovered earlier if we had verified that responses to heartbeat packets were only a few dozen bytes.
- Is different data corrupted in unusual ways in response to specific input? For instance, does a longer input cause a different failure than a shorter input? This suggests a write BOF.
- Does the program crash, and a dump or debugger give nonsensical stack traces? This suggests a write BOF of stack locations that corrupts the call/return stack.

Static analyzers check programs for possible BOF and other issues. Sound static analyzers are potentially always correct. In contrast, heuristic analyzers generally run faster, handle more languages, and cover more classes of vulnerabilities. Today, most static analyzers have lower false-positive rates and simultaneously lower false-negative rates than they had in the past. Some static analyzers have been augmented with execution monitoring to yield hybrid (static and dynamic) analyzers.

Good general testing techniques complement static analysis. Testing relies on fewer assumptions and checks properties that are difficult to specify. We mention a few points particularly important to testing for BOF:

- Try to exceed limits, check routines that allocate more memory, and challenge the limits of hard-coded arrays.
- Try very unusual inputs, such as negative numbers, empty fields, and letters or special symbols where numbers are expected.

In contrast to the aforementioned external methods, internal detection mechanisms have access to the program's state and control flow. Many of them not only detect BOF but also help prevent failures or lessen their impact. Therefore, in the next section, we include internal ways to mitigate or preclude BOF with the discussion of ways to internally detect them.

Internal Detection and Prevention

The best way to prevent BOF is to reduce the use of C. Optimizing compilers and multicore processors removes most concerns about slower execution, allowing programmers to work on algorithmic improvements instead of checking every array access for a possible BOF. If you must write in C, use more structured stores, such as associative memory, prop lists, graphs, queues, sets, stacks, or trees. These abstract data structures bundle accessing operations that only allow access to valid elements. Arrays have minimal structure: just the index order.

There are many internal techniques for detecting, mitigating, or precluding BOF faults. They are either passive (detect that BOF has occurred) or active (prevent BOF). Also, they either require a programmer's action in order to be inserted or are inserted automatically by the compiler or the OS. One technique is to add checks to verify that every access is within bounds. Research shows that many bounds-checking tools or libraries have little impact on speed.[2] Chips with multiple, deeply pipelined cores can check bounds while the array is being accessed. Checking can also be done by the hardware. For instance, arrays might have read-only or unallocated blocks of memory on both ends. Small invalid array accesses result in memory violation interrupts. If performance still suffers, such checking could be enabled during development and testing, then disabled for production.

Some of these techniques might not be applicable—for example, if the size of the buffer is not available to check. In such cases, more sophisticated techniques attempt to foil adversaries.

Shadowing and fat pointers keep additional information about memory use and allocation in other parts of memory to enable access and taint checks.[3] Address space layout randomization (ASLR) distributes arrays unsystematically in memory. With ASLR, a BOF is unlikely to access the same unassociated object in different executions without a lot of work. Information that is connected, such as in the stack or in the same structure, is harder to rearrange. Padding allocates extra space for every array, so small magnitude BOF events might not cause problems. “Canaries” are special values, such as 0xDEADBEEF, added before and after arrays. If these values are changed, it is likely that a write BOF occurred.

Testing for Buffer Overflows

Testing for BOF is still crucial even when programs use good techniques. Test cases specifically targeted to exposing BOF can be generated through fuzzing, memory checking, and negative testing.

Fuzzing is a class of techniques in which random or structured random input is presented to a program with only limited checking of the outcome. Often, the only checking is that the program did not crash or hang. Because fuzzing automates input generation and output checking, huge numbers of tests can be run at little cost other than a few hours of computer time. Fuzz testing gets its power because random inputs expose the limits of programmers' analyses, or they violate assumptions about inputs that can never occur.

Structured random inputs are more powerful than purely random inputs, given that the latter primarily exercise the input checking routines. For instance, if a particular input is a date, it is useful to run only a moderate number of purely random tests. Any additional random tests are almost always handled by the code that tests whether the date is invalid. After a moderate number of tests, structured random dates can be generated with random months from 1 to 12, random days from 1 to 31, and a wide range of years. Another approach to structured random inputs is to capture known input and randomly mutate it. For instance, image display programs can be fed actual images with random changes.

Fuzzing with memory checking can be very effective. For instance, *american fuzzy lop* (afl) “tracks the branches that are taken and how often, then prefers using tests that cover the program differently when it evolves new tests” (http://lcamtuf.coredump.cx/afl).

Exact memory checkers, such as *Address Sanitizer* (ASan) or *Purify*, check memory allocation and layout. The overhead can be significant, up to twice the execution time and memory use, but this may be cheap insurance against vulnerabilities.

Negative testing examines how the program behaves when inputs are not as expected. The vast majority of testing is designed to gain confidence that the program produces expected outputs for typical inputs. As Wheeler says,

> Thorough negative testing … creates a set of tests that cover every type of input that should fail. … This would have immediately found Heartbleed, since Heartbleed involved a data length value that was not correct according to the specification. It would also find other problems like CVE-2014-1266, the goto fail error in the Apple iOS implementation of SSL/TLS.[4]

You do not have to suffer from BOF. Buffer overflows can cause serious problems, especially when we acknowledge the possibility of adversaries who try to exploit vulnerabilities in your programs. The best approach to ensuring that your software does not have buffer overflows is to use a programming language in which such bugs are impossible (memory access is always handled reliably) or, at least, can surely be detected by tools during production. There are many techniques that detect the vast majority of BOF. There is no reason for your development process to be interrupted, scrambling to patch them.

References

1. Bojanova I, et al. The Bugs Framework (BF): A Structured Approach to Express Bugs. Proc 2016 IEEE Int'l Conf Software Quality, Reliability, and Security, to appear. 2016
2. Flater, D. “Defensive Code's Impact on Software Performance,” tech note 1860, US Nat'l Inst. Standards and Technology. Jan. 2015 https://doi.org/10.6028/NIST.TN.1860
3. Berger, ED., Zorn, BG. DieHard: Probabilistic Memory Safety for Unsafe Languages; Proc 27th ACM SIGPLAN Conf Programming Language Design and Implementation. 2006. p. 158-168.https://people.cs.umass.edu/~emery/pubs/fp014-berger.pdf
4. Wheeler, DA. How to Prevent the Next Heartbleed. blog. Apr 29. 2014 www.dwheeler.com/essays/heartbleed.html

Biographies

Paul E. Black is a computer scientist at the US National Institute of Standards and Technology. His research interests include static analysis, software testing, networks and queuing analysis, formal methods, software verification, quantum computing, and computer forensics. Black has nearly 20 years of industrial experience developing software for integrated circuit design and verification, assuring software quality, and managing business data processing. He is the founder and editor of the Dictionary of Algorithms and Data Structures (www.nist.gov/dads/), and is a member of ACM and a senior member of IEEE. Contact him at paul.black@nist.gov.

Irena Bojanova is a computer scientist at the US National Institute of Standards and Technology. She serves as the Committee on Integrity chair of the IEEE CS publications board, an associate editor in chief of IT Professional, cochair of the IEEE Reliability Society's Technical Committee on the Internet of Things, and a founding member of the IEEE **Special Technical Committee** on Big Data. She was the founding chair of the IEEE CS Special Technical Community on Cloud Computing andthe editor in chief of Transactions on Cloud Computing. You can read her blogs, “Sensing IoT” and “A Cloud Blog,” on Computing Now. Contact her at irena.bojanova@computer.org.

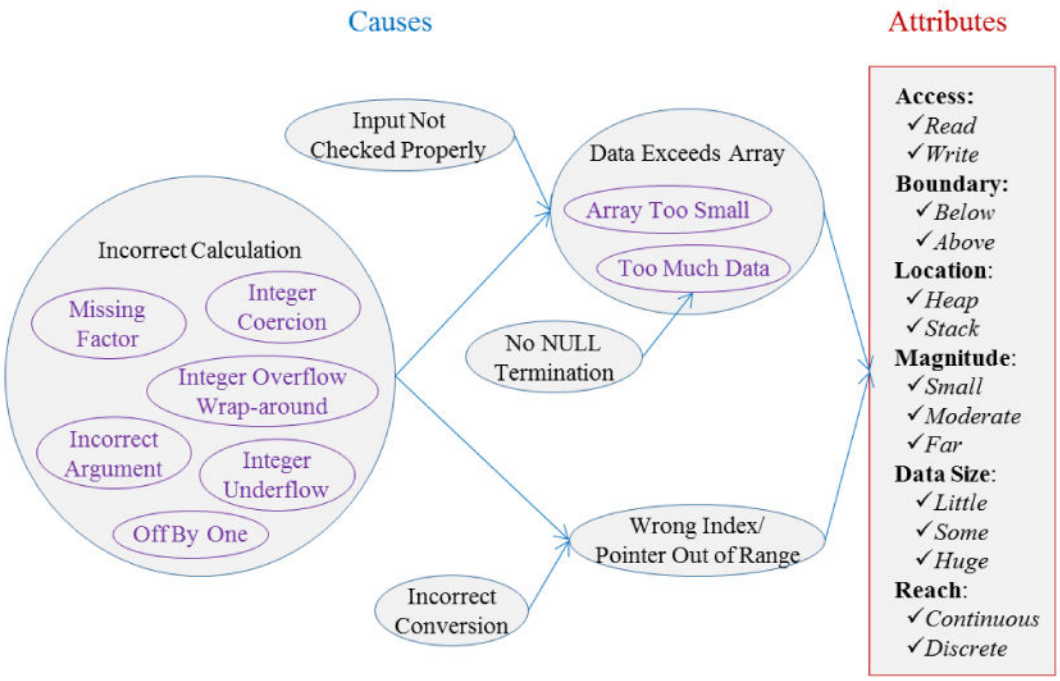

Figure 1. Buffer overflow (BOF) causes and attributes[1]