Editors: Richard Ford, rford@se.fit.edu | Deborah A. Frincke, debfrincke@gmail.com

Heartbleed 101

Marco Carvalho | Florida Institute of Technology
Jared DeMott | Bromium
Richard Ford | Florida Institute of Technology
David A. Wheeler | Institute for Defense Analyses

Described by some as the worst vulnerability since e-commerce began on the Internet, one word sums up what this Basic Training column is all about: Heartbleed. Although we don't necessarily agree with such hyperbole (although it really was pretty bad!), the media furor around the Heartbleed vulnerability was incredible and crossed over from security mailing lists to the national press with remarkable speed. Here, we take a look at this vulnerability in OpenSSL and outline how it was fixed. Perhaps more important, we also step back and look at the issue more broadly. Why was the Heartbleed vulnerability missed for so long?

Basic Anatomy

To understand Heartbleed, you must first understand the functionality that OpenSSL developers were trying to enable, which takes us all the way back to RFC 6520, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension." This seemingly benign document described an extension to the TLS protocol designed to enable a low-cost, keep-alive mechanism for peers to know that they're still connected and all is well at the TLS layer.

The implementation of RFC 6520 was, in theory, simple: send a packet of type `heartbeat_request`, along with an arbitrary payload and a field that defines the payload length. The request should be answered by a response that contains an exact copy of the payload. This mechanism would allow for more streamlined checking of connection state and lower client/server overhead on long-lived connections. Unfortunately, as history shows, a lot can go wrong between design and implementation.

Version 1.0.1 of OpenSSL added support for the Heartbeat functionality and enabled it by default, thereby inadvertently making the implementation vulnerable by default. This vulnerability remained until 1.0.1g—or roughly two years.

The actual source code patch to fix Heartbleed is shown in part in Figure 1 and is informative in its simplicity. In essence, the code merely adds a check to make sure that the response isn't longer than the request in Figure 1.

This error existed in the source tree in a couple of places, and the fix is slightly longer than shown, but it really does capture the gist of it. Before this patch was added, the `heartbeat_request` packet essentially blindly returned a block of memory corresponding to the package's stated payload size instead of its actual size. With many thanks to xkcd's Randall Munroe, Figure 2 shows the Heartbleed vulnerability graphically (http://xkcd.com/1354). Yes, it really is that simple.

When we look at Heartbleed on the surface, we see that it's essentially a buffer overread vulnerability, where inadequate bounds-checking is carried out at runtime. Indeed, the fix addresses precisely this problem and carefully checks that the read doesn't include data unrelated to the request.

At a slightly more sophisticated level, another way of looking at this is that the programmer placed trust in something that was utterly untrustworthy and not dependable. A client cannot and should not trust the payload length presented in the `heartbeat_request` packet. At a more philosophical level, this was the root cause of the vulnerability, but it doesn't apply to just bounds checking: placing trust in user-supplied input is often a bad idea.

This kind of vulnerability has historically occurred several times in early Web applications, when the server couldn't rely on the client to provide trustworthy responses. For example, a client-side JavaScript-controlled dropdown menu doesn't actually constrain input from an untrusted client, and an unsigned cookie stored on a client should never directly determine the value of items stored in a shopping cart (a vulnerability that existed in an early

1540-7993/14/$31.00 © 2014 IEEE Copublished by the IEEE Computer and Reliability Societies

```
+ /* 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 */
+ pl = p;
+
```

Figure 1. Part of the source code path to fix Heartbleed.

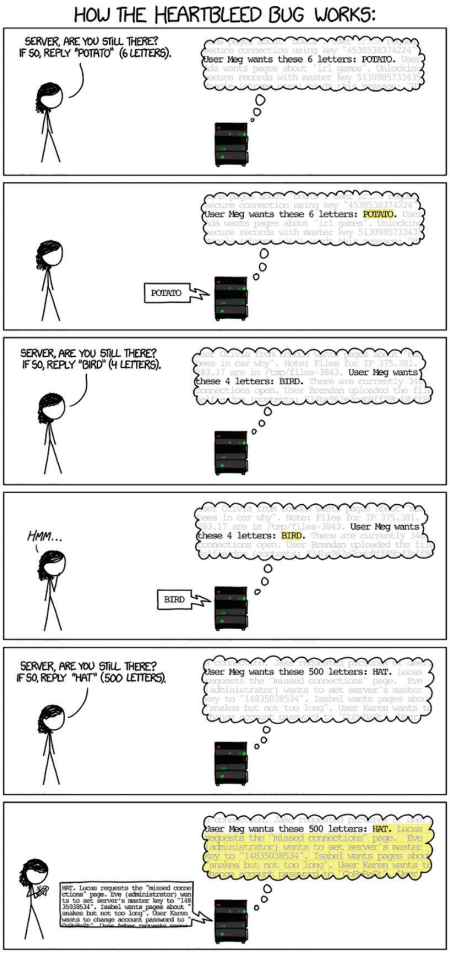

Figure 2. The separation provided by the seL4 microkernel. This separation lets us build well-performing systems with millions of lines of legacy code, while reducing the trusted code base to a manageable level. (Source: XKCD, used with permission.)

e-commerce system). In security, keeping track of what's trustworthy and what's tainted and unreliable is key. This mindset would have prevented the programming error found in the code.

"What was Heartbleed? How was it exploited? How was it fixed?"—all of these are easy questions. What's much more interesting is exploring the factors that allowed such a serious bug in a security-sensitive and prevalent software component to exist for so long.

Many Eyeballs, Shallow Bugs?

One of the most well-known quotes about the open source movement comes from the *Cathedral and the Bazaar*, by Eric S. Raymond. It's all too common to hear people throw "Given enough eyeballs, all bugs are shallow" around as if it tells the whole story about the trustworthiness of open source code. Unsurprisingly, the discussion in Raymond's book is much more nuanced, but the misconception that open source is somehow magically protected by "the community" has become common—at least among those who aren't actively involved in the tough world of developing open source software. This false impression can be counterproductive, because security testing requires directed effort, regardless of the license or development methodology used.

With that said, the fact remains that it took a significant amount of time for this flaw to be discovered. Why wasn't it found with existing automated testing techniques? Let's examine the two best-known approaches: static analysis of source code and input fuzzing.

Static Analysis

Static analysis involves examining software without executing it, and it includes anything from human review to a variety of automated tools. One of the most common types of static analysis tools is the source code weakness analyzer (also known as the static application security testing tool), which examines a program's source code to find potential vulnerabilities. But as far as we know, all existing source code weakness analyzers wouldn't have found this vulnerability by default, including those by Coverity, HP/Fortify, Klocwork, and GrammaTech.

It's important to understand that these analyzers don't verify the absence of vulnerabilities but instead try to find as many as possible. In short, they're incomplete. This is intentional; most programming languages aren't designed to be easy to analyze, and most software isn't written to make it easy for static analyzers to analyze. Complete analysis tools often require a lot of human help to apply to existing programs. In contrast, incomplete analysis tools can be applied immediately to existing programs by using various heuristics. However, this presents a major caveat: incomplete source code weakness analyzers often miss vulnerabilities. A partial solution is to use multiple tools; that way, if one tool misses the vulnerability, another tool may find it.

In this case, though, OpenSSL's complex organization exceeded the ability of all of these tools to find the vulnerability. James Kupsch and Barton Miller identified four factors that made OpenSSL especially difficult to analyze (https://continuousassurance.org/swamp/SWAMP-Heartbleed.pdf):

- Use of pointers. Pointers make analyzing memory use difficult, "because the size of the buffer is not contained in the pointer but must be stored and managed separately from the pointer." This makes it difficult for a tool to track the size of an object being pointed to. Heartbleed had multiple levels of indirection that made it especially difficult for a tool to track what was valid (the vulnerable function was passed a pointer to a structure containing another pointer to a structure with a field that points to the actual record). OpenSSL also uses function pointers to provide extra flexibility, yet function pointers are especially difficult for tools to handle.
- Complexity of the execution path from buffer allocation misuse. There's no necessary direct code path from memory allocation, through writing, to its invalid use. Instead, buffers are cached and reused, which can aid performance but make it exceedingly difficult for tools to identify invalid uses.
- Valid bytes of the TLS message are a subset of the allocated buffer. As Kupsch and Miller state, "The pointers to the message and the payload both point into the middle of a buffer, and the contents of the message do not use the entire memory buffer. For a tool to track the correct memory usage in a situation like this, a tool needs to track the boundaries of the object and the containing buffer.... The length of the message is much more difficult [to determine] as it depends on the semantics of the program."
- Contents of the buffer don't appear to come directly from the attacker. Many static analysis tools perform "taint analysis." They mark data from untrusted sources as tainted and limit how it can be used. Most tools then use heuristics to determine when the data becomes untainted. However, the custom memory allocators make it difficult to determine which data is tainted and which is not. In addition, the process of decrypting, uncompressing, and verifying message integrity looks similar enough to data validation that tools may consider it untainted.

You can improve the results of a source code weakness analyzer by providing detailed information about the program that you're analyzing. We call this approach a "context-configured source code weakness analyzer." Klocwork has shown, for example, that Heartbleed could have been found if additional information about OpenSSL was provided (https://continuousassurance.org/swamp/SWAMP-Heartbleed.pdf). This does require, however, more effort.

Source code weakness analyzer developers continuously improve their tools, especially when important vulnerabilities like Heartbleed are missed. Coverity, for example, recently developed some new heuristics that it thinks would've detected Heartbleed (http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html). New heuristics will never lead to finding all vulnerabilities, but they can help find more.

Fuzzing

Given the limitations of static analysis, let's turn our attention to another security-testing technique, which many feel could have found the vulnerability quickly and easily. Fuzzing is a security-focused testing approach in which a compiled program is executed so that the attack surface can be tested as it actually runs. Typically, attack surfaces are the components of code that accept user input. Because this is the most vulnerable part of code, it should be rigorously tested with anomalous data. During testing, the application is monitored for known bad states, such as an application crash, often the result of an out-of-bounds memory access. If a crash is found, the input and application state are stored for later review. Such a flaw will be entered as a security bug for developers to repair. Figure 3 shows a high-level view of fuzzing.

Fuzzing is often an effective way to find bugs missed in manual and automated code reviews. Fuzzing also finds real flaws and has a very low false-positive rate when compared to techniques such as static analysis. However, fuzzing tends to get shallow penetration for complex protocols, and thus has relatively weak code coverage. This is particularly true for code paths that might require the specialized input unlikely to be present in automated inputs. This is why both static and dynamic testing approaches are critical to any well-formed secure development life cycle.

While the Heartbleed wiki page specifies how and when the bug was introduced into the code base, it doesn't disclose how the OpenSSL code was security tested, either statically or at runtime. It wouldn't be surprising to find out that a vulnerable version of the OpenSSL code had been fuzzed and the Heartbleed bug had been missed. Information disclosure bugs are easy to miss when fuzzing: there might not be a crash associated with this bug. Heartbleed is an overread bug, not an overwrite bug, so many fuzzing setups simply wouldn't catch it.

However, crashes in the default environment aren't the only exceptional condition fuzzers can look for. Fuzzing tools can also observe potential memory leaks—for example, for a network protocol such as OpenSSL, the size of return packets could be recorded, and anything larger than expected should be reported on. Likewise, if the returned data is of a well-

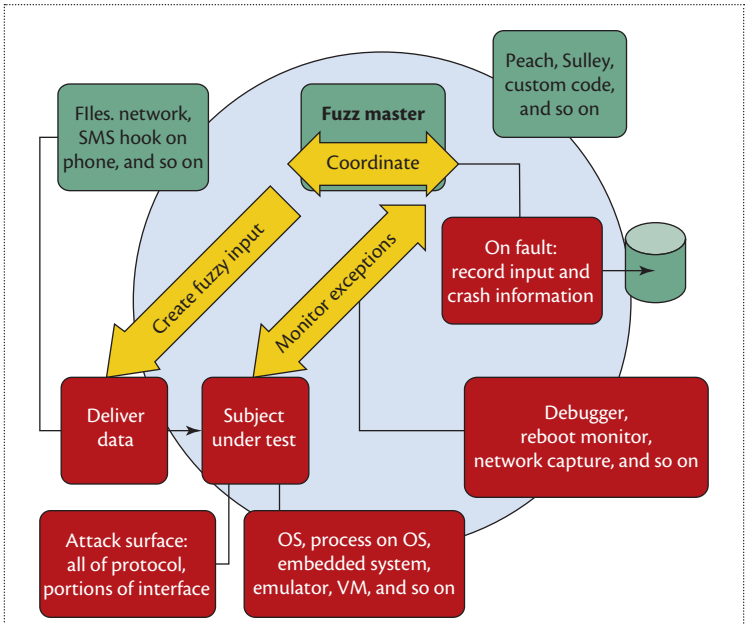

Figure 3. Fuzzing overview. Malformed input is created and sent to a program, which is being monitored by a debugger, with results being collected if a crash is found.

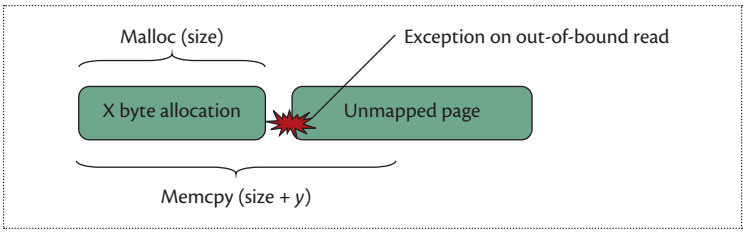

Figure 4. Debug heap. A special memory allocator can place each valid memory allocation next to an invalid allocation so that overreads or overwrites of even a single byte are caught instantly.

known format, unknown data should also be treated with suspicion. Detecting leaked memory is commonly accomplished by using a debug heap that sets each allocation next to an unmapped page, as Figure 4 shows.

If data is read beyond the scope of one allocation, a page fault will occur. Not all information leaks occur outside the bounds of an allocated buffer, but this approach is a good start. Most operating systems have a debugging system available with a debug heap that can be optionally enabled. The heartbeat extension wasn't fuzzed using a debug heap, or else this bug would have been detected. The beauty of using debug heaps for fuzzing is that your existing tools can continue to focus on catching crashes, and they'll now find this bug type. The tradeoff is that the application, and therefore the fuzzer, will run much slower and require much more memory.

It's impossible to know what security tools were or weren't used to analyze OpenSSL for vulnerabilities; people rarely report unsuccessful attempts. The real point is that many of the tools commonly used would not—or might not, unless very carefully applied—have found the Heartbleed vulnerability.

Looking to the Future

Post Heartbleed, perhaps the best result we can hope for is that development organizations will examine why current approaches failed and apply additional approaches that will counter Heartbleed-like vulnerabilities in the future (for example, see www.dwheeler.com/essays/heartbleed.html). In many cases, this might be hampered by the limited support that some open source projects receive in terms of funding. OpenSSL is used very broadly and is an important part of the security ecosystem, yet *The New York Times* has reported that the project has a typical budget of just US$2,000 a year (www.nytimes.com/2014/04/19/technology/heartbleed-highlights-a-contradiction-in-the-web.html?_r=0#). Despite the heavy commercial usage of many open source projects, for the programmers involved, it's sometimes just a labor of love.

In terms of technical preventives, static analysis and input fuzzing are some of the techniques normally used to find bugs or vulnerabilities in code and are intrinsically limited to the scope that they're designed to cover. Even if they were applied in this instance, they would have missed other serious vulnerabilities in the complex software systems that we use today. This is essentially the nature of the game, and while our current techniques are insufficient, they are and will continue to be an important and necessary part of the process of building more secure software systems. Improvements and additions to these techniques have been suggested (www.dwheeler.com/essays/heartbleed.html), but we shouldn't expect such approaches to be perfect.

Thus, while it is indeed necessary to continue to advance and improve our security testing tools and tech-

niques, we must also recognize their limitations and look into alternative or complementary ways to defend our systems and information. In recent years, the notions of dynamic and moving-target defenses, for example, have been proposed as ways to improve the resilience of software systems. These techniques propose a changing attack surface that would, at least in theory, make it more difficult or costly for an attacker to identify and exploit a given vulnerability. Dynamic attack surfaces are normally achieved through runtime and periodic changes in system configuration, ranging from low-level address layout to communication protocols, operating systems, and service implementations. Ironically, one of the most commonly used moving-target defenses (address space layout randomization) wouldn't have helped reduce the impact of the Heartbleed vulnerability.

Would any dynamic defense or moving-target technique have helped prevent Heartbleed? It's still too early to say if they would have made a significant enough difference to justify their costs. However, it isn't unreasonable to consider a diverse implementation that could somehow be coordinated to correlate inputs and responses. Such an approach might have exposed a larger attack surface but could also have helped provide a much earlier detection of the vulnerability's exploitation.

The far future might depend not on any one specific technique but on the appropriate use and coordination of multiple approaches, tools, and techniques. ■

Marco Carvalho is an associate professor of computer sciences and the director of the Harris Institute for Assured Information at the Florida Institute of Technology. Contact him at mcarvalho@fit.edu.

Jared DeMott is a principle security researcher at Bromium. Contact him at jdemott@vdalabs.com.

Richard Ford is the Harris Professor of computer sciences and department head for the Computer Sciences and Cybersecurity Department at the Florida Institute of Technology. Contact him at rford@se.fit.edu.

David A. Wheeler is a research staff member at the Institute for Defense Analyses. Contact him at dwheeler@dwheeler.com.