Why Do Software Assurance Tools Have Problems Finding Bugs Like Heartbleed?

James A. Kupsch and Barton P. Miller

Software Assurance Marketplace (SWAMP)
University of Wisconsin

The nature of the Heartbleed vulnerability [1] has been well described by several people including Matthew Green [2], Chris Williams [3], Troy Hunt [4], Eric Limer [5], and Sean Cassidy [6]. We appreciate their clear descriptions of the vulnerability, the code, and its consequences. We want to take a different slant: what makes the Heartbleed vulnerability difficult for automated tools to discover? First, we provide our thoughts on why both static and dynamic assurance tools have a difficult time discovering the vulnerability and why current defensive run-time measures were not successful. Second, we discuss how in the future the SWAMP (Software Assurance Marketplace) could serve as a valuable resource for tool researchers and developers to facilitate the discovery of weaknesses that are not found by existing tools and to facilitate testing of new innovative tools.

Heartbleed created a significant challenge for current software assurance tools, and we do not know of any such tools that were able to discover the Heartbleed vulnerability at the time of announcement. The thing to remember is that this is one bug in one program whose structure made the discovery of this bug particularly difficult. Software assurance tools do find many real vulnerabilities. Coverity recently announced [7] an experimental checker that finds Heartbleed but is still undergoing testing with other code bases to verify that it generalizes and has an acceptable false positive rate. By analyzing a bug as Coverity has done, tools can be improved to detect previously undiscoverable bugs.

1. Background on Heartbleed

Heartbleed is a vulnerability in the OpenSSL [8] implementation of the TLS [9] and DTLS [10] protocols (successors to SSL), specifically in the heartbeat extension [11]. The vulnerability manifests itself in a similar manner in both TLS and DTLS, so our comments about TLS apply equally to DTLS. TLS enables secure communications between two processes residing on different hosts, most importantly used to secure web and e-mail. The vulnerability can be exploited to leak confidential information from inside a process using the OpenSSL library to another process when the two are communicating using the TLS protocol. The leaked information can include usernames, passwords, emails, confidential information entered or displayed on a web page, and authentication data, including secrets keys used to protect all traffic connecting to the host.

The crux of the problem is the failure of the OpenSSL library to validate two bytes of data [12], a length field, received in a heartbeat request message that is processed by the tls1_process_heartbeat function, partially shown below in Listing 1. The missing validation is shown in the patch that fixes the vulnerability [13], shown in Listing 2. The heartbeat protocol is supposed to echo back the data sent in the request where the amount is given by the length field. An omniscient tool would know that 1) the memory for the heartbeat request message (variable "p" on line 2556 of Listing 1) is supplied by an untrusted source, 2) the length field (variable "payload" assigned in line 2563 of Listing 1) is therefore also untrusted, 3) the beginning of the heartbeat payload is stored in variable "p1", 4) the length of the containing TLS record is given by "s->s3_rrec.length", 5) the length of the payload is constrained to a value of 0 to the TLS record length minus 19, 6) the heartbeat response message is being created in the lines 2580 to 2589, 7) the payload of the heartbeat response message should be a copy of the payload of the heartbeat request message, and finally (performed in line 2586 of Listing 1 using the memcpy function), 8) nothing enforces this constraint, allowing the memcpy in line 2586 of Listing 1 to read memory that should not be read up to 64KB [14] and to exfiltrate it back to an attacker in line 2591 of Listing 1. Omniscient tools do not exist, and tools do not have the reasoning to discern these facts. Instead, they operate on an abstraction of properties of the programming language, libraries, commonly used idioms (both good and bad), heuristics, a logic about how new facts are derived, and what evidence constitutes a safety violation. Since the abstraction and analysis are simplifications, real problems are missed and false problems are reported, but by improving the abstractions and analysis, the problems can be reduced.

2. Static Analysis Tools

Static Code Analysis tools are software assurance tools that look for problems by examining the source code of a program. These tools can find questionable use of user supplied data and accessing the contents of an array outside its bounds. What properties of the OpenSSL library made it difficult for an automated tool to discover? We postulate this difficulty comes from four main sources: 1) the use of pointers, 2) the complexity of the execution path from the buffer allocation to its misuse, 3) the valid bytes of the TLS message are a subset of the allocated buffer, and 4) the contents of the buffer do not appear to come directly from the attacker. Each of these is discussed next.

(a) Use of pointers

The use of pointers makes analysis of the correct use of memory difficult, because the size of the buffer is not contained in the pointer but must be stored and managed separately from the pointer. In C, the language in which OpenSSL is written, there are two ways to create a buffer, statically declare an array variable in the source code (The size is determined at compile time and is relatively easy for a tool to analyze.) or declare a pointer to a memory address and then assign as its value a buffer allocated with a run-time computed size. In C, a pointer can point to the memory of statically allocated variables or to dynamically allocated buffers. The memory pointed to by a pointer can be deallocated or modified multiple times during the execution of a program. All of these unknowns conspire to make it difficult for an automated tool (or even human analyst) to determine the properties of the memory pointed to by the pointer.

In the case of the Heartbleed vulnerability, there is a pointer to an output buffer where the response is created (variable "buffer" on line 2580 of Listing 1). This buffer is dynamically allocated in the tls1_process_heartbeat function. Since this buffer is allocated, used, and freed in the same function, this pointer is relatively easy for a tool to analyze. There are no problems with the use of this pointer. A second pointer points into the input buffer that contains the heartbeat record. The value assigned to the second pointer (variable "p" on line 2556 of Listing 1) is a pointer into a dynamically allocated buffer. This is the buffer that is improperly used in the Heartbleed vulnerability.

Worse yet, the data structure passed to the function processing the heartbeat record is passed as a pointer to a structure containing a pointer to a structure that contains fields that point to the heartbeat record. (The fields are a pair which include a pointer to the buffer containing the record and the length.) All these levels of indirection make it unlikely that a tool is able to keep track of the values used to create the buffer and how many bytes of the buffer are valid.

Another use of pointers in the OpenSSL library that makes analyzing the code difficult is the use of function pointers. To provide flexibility in the use of the OpenSSL library, the actual function to call is determined at run-time by configuration file or negotiation with the peer. This greatly increases the complexity of the analysis since the tool must consider all possible combinations of all possible flows through the function, including those that are not possible in actual use, as the tools do not analyze the configuration file or understand the restrictions in the protocol. The space of possible flows is near infinite for a library like OpenSSL so tools must make compromises in their analysis. Some tools will not consider following function pointers, while others make simplifying assumptions that do not accurately reflect what the code actually does. All of this results in a reduction of fidelity of the analysis.

(b) Complexity of the execution path from buffer allocation to misuse

The input buffer for the TLS session is dynamically allocated when a TLS session is first started and is normally reused during the duration of the TLS session and not released until the session is closed. Another mode of operation creates a pool of buffers that are reused between TLS sessions. In either case, a buffer likely comes from a pool of previously allocated buffers for sending or receiving TLS data. There is not a direct path (a chain of function calls where data is passed from function to function) through the code from the initial allocation of the memory to the writing of the memory and finally to the invalid use that occurs in tls1_process_heartbeat. The path depends on the number of TLS connections, their order, the calls into the OpenSSL library, and the order of packets arriving from the network.

Tracking all of the permissible paths through the code and keeping track of all the values for the variable makes it infeasible to do perfectly or even allowing for a small number of false positive results.

(c) Valid bytes of the TLS message are a subset of the allocated buffer

For each session, there is an input buffer where data received from the network is placed as it is read. The TLS message is contained in part of this buffer. The buffer is large enough to be able to contain the largest permissible TLS record (16KB plus 1 to 2KB for overhead). This buffer is also used to decompress and decrypt incoming data in place.

The TLS protocol is a layered protocol where different parts of the protocol can have variable lengths. The outermost layer is called a TLS record, and there are five types of records, the heartbeat record being one of these types. A TLS record consists of five bytes of header (type, version, and length) and the rest of the data being a single message or sequence of messages of the given record type. The OpenSSL library reads one record into the input buffer using the length field of the header to determine the amount of data to read. The rest of the record is usually encrypted, with an integrity checksum, and is possibly compressed. The library decrypts, uncompresses, and verifies the message and, if valid, calls a function that is specific to the record type that processes the messages contained within the record.

In the case of a heartbeat record, it contains exactly one heartbeat message. The message contains four fields (type, length of the payload, payload, and padding). The type denotes a request or a response. The length field denotes the length of the payload, while the length of the padding is determined by subtracting the lengths of the other fields from the total length of the other fields and must be a minimum of 16 bytes. These restrictions result in the validation code shown in Listing 2.

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. This requires tracking four values: a pointer to the message and its size, and a pointer to the containing buffer and its size. Three of the values, at least in simple code and when using standard allocators, are straightforward to compute: the pointer to the message, the pointer to the buffer, and the buffer's size. The length of the message is much more difficult as it depends on the semantics of the program. A trivial bound on the length of the message is the end of the containing buffer, but a more precise constraint is very difficult to determine. Without knowing the length of the message, accessing data beyond the end of the message will go undetected. Knowing the size of the containing buffer would have potentially produced a warning in the case of Heartbleed since the containing buffer is smaller than the range of the length field.

(d) Contents of the buffer do not appear to come directly from the attacker

A common technique used by static analysis tools is to perform taint analysis where data that can come from an untrusted source, such as the network, is marked as being tainted and produces violations if tainted data is used in potentially unsafe manners. Then using dataflow, the taintedness can be propagated to other values in the code. The problem for the tool then becomes how the data becomes untainted. For a general purpose tool, it places a large burden on the user if they manually have to declare that a value that would be otherwise tainted is no longer tainted. To reduce the burden, most tools use heuristics to determine when data becomes untainted. Usually the heuristic involves all the bytes of the data being tested or used in a complex calculation to create a new value.

For a tool that does taint analysis of data, if the tool was sophisticated enough to propagate the taint analysis through the multiple pointers, then the process of decrypting, uncompressing, and verifying the integrity of the message may look enough like validation of the data to mark the data as untainted.

A sophisticated tool might be able to point to tls1_process_heartbeat as using the read buffer outside its bounds but would likely only point to the length field being unbounded and the buffer having a size of 17-18KB. The use of the custom allocator would need to be removed to make this possible. The bug would still be undetectable if a check improperly restricted the data to 17-18KB as the input buffer is that size. Annotations in the code to link relationships between pointers and variables that point inside the same buffer or contain the length might enable tools to detect more defects.

3. Dynamic Analysis Tools

Dynamic analysis tools operate by monitoring a running application for violations of safety. The dynamic tool has to be run and driven with test data, and that test data has to trigger the violation if it is present in the code. This is the difficulty with dynamic assessment tools; the results are only as good as the test data used to drive them. For bugs that occur during normal operations using unmodified code, dynamic tools can be quite useful.

One family of dynamic analysis tools detects unsafe use of memory. A tool such as this is ValGrind [15], which can detect access to uninitialized memory or to memory outside of a properly allocated memory.

It appears that this type of tool should have been able to find the Heartbleed vulnerability at the time that memory is accessed outside the valid buffer range, but it does not for a couple of reasons.

First, the Heartbleed bug is not triggered by a client using an unmodified (correctly operating) TLS library as TLS implementations never send a malformed heartbeat message. To use a dynamic tool would first require that a modified library be created with test data that produces the malformed packet. This is a non-trivial task. One method that might prove fruitful to find Heartbleed and similar problems would be to create hooks to mutate the message data using a form of fuzz testing before it is encrypted and sent to the other side. This fuzz testing has the potential to cause the other end of the TLS connection to return records that are invalid. In the case of a heartbeat, the record would contain a message that is too large, and the size of the response would be different than the size sent.

Second, tools will not automatically recognize the use of non-standard calls that allocate or release memory, and OpenSSL uses a custom memory allocator. To a dynamic analysis tool, it appears as if the library is allocating large memory buffers and not returning them, but in reality, it is subdividing these large blocks of memory and returning them for use. This allows the Heartbleed reads, which span multiple buffers, to go undetected, because they appear to be from same large buffer.

Dynamic tools hold promise in detecting this type of problem with the one time expense of writing a custom library that can act as a fuzzing engine. The custom allocator would also have to be disabled, and the errors would present themselves faster if buffer reuse was disabled and if buffers were allocated to the exact size necessary.

4. Protective Run-time Measures

One protective run-time measure that could have detected Heartbleed or at least restricted the amount of memory that could have been exfiltrated is a memory allocator that uses guard pages around allocated memory blocks and unmaps memory that has been freed. Attempted access to a guard page or freed memory results in fault that causes the program to terminate. Also, when a buffer is allocated, the content of the memory is zeroed, which also reduces the risk of leaking data. This type of protection is common in modern heap management code.

OpenSSL's custom memory allocator, as described above, prevents the defensive memory allocator from being useful as a defensive countermeasure. Theo de Raadt from the OpenBSD project and another developer commented on exactly this fact [16].

5. The SWAMP as a Resource for Tool Research and Development

The SWAMP provides a facility that can help researchers and developers improve their software assurance tools. Some of the key services provided by the SWAMP are its collection of reference data sets (programs with known weaknesses and vulnerabilities) and the ability to continuously run multiple tools side-by-side, comparing their results.

Right now, the SWAMP provides a corpus of about 350 diverse software packages and test suites. Many of these are known to have known weaknesses or vulnerabilities. In the future, the SWAMP will greatly increase this corpus of software packages. The SWAMP automates the application of a tool against any or all of the packages in the SWAMP. The same can be done with any of the tools in the SWAMP and results viewed together. This allows a tool developer to discover defects found by other tools, but not their own tool. The unified view of tools provides the basis for tool developers to identify gaps in their tools.

This insight can be valuable for tool developers as they are now able to create techniques to improve the coverage areas in their own tool.

In addition, the SWAMP will add software packages, along with a description of known vulnerabilities within the software *and* locations known to be vulnerability-free (places that were previously known to be reported as a false positive). An initial version of the data for the bug locations and description could be gathered from bug databases and source code repositories or by manual entry. Another source of data is by running the tools that are part of the SWAMP and making those results available. The quality of this data could then be improved using human analyst and crowd sourcing techniques to triage the results. Over time, this will refine the results, increasing the quality.

With respect to new and serious vulnerabilities such as Heartbleed, the SWAMP has several things to offer. First, we can easily incorporate new tools and new types of tools. For example, OpenSSL has been criticized as to some of the programming style and engineering choices that were made (or that evolved). A software engineering tool that evaluated style, as opposed to particular weaknesses or vulnerabilities, could help the designers simplify their code, making it less likely to have obscure weaknesses and more easily let tools detect such weaknesses.

Second, we can quickly incorporate new test cases. For example, we are working to produce a simplified version of the Heartbleed vulnerability and present that as a software package in the SWAMP. Such a package provides an easy-to-use test case for tools developing new heuristics for such cases and for tools taking on the greater challenge of first principles attacks on such weaknesses.

Third, we can allow an open comparison and combination of a variety of types of tools, including static and dynamic code analysis, source and binary, and run-time monitoring. The goal is to provide the best tool or combination of tools for each situation.

The SWAMP is designed to help facilitate technical exchange and research collaboration in the software assurance community that will help enable new discoveries, improvements, and advancements in software development and software assurance activities. As new attacks emerge, new defenses will be needed in response to these attacks. The SWAMP's goal is to try to stay ahead of (or at least keep pace with) the bad guys, providing key resources and services to the software assurance community to improve and maintain the quality of software.

Acknowledgements

We gratefully acknowledge the comments and suggestions from Miron Livny, Von Welch, and Kevin Greene. These comments helped to shape the discussion of the role of the SWAMP project with respect to the software assurance community's response to the Heartbleed vulnerability. We also thank Tom Ristenpart (Assistant Professor at the University of Wisconsin Computer Sciences Department) for acting as a sounding board

and providing valuable comments on the software engineering issues of OpenSSL as they relate to Heartbleed.

Listing 1. tls1_process_heartbeat function containing Heartbleed vulnerability from OpenSSL version 1.0.1f

```
tls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
        {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 bytes
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);

        r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
                             3 + payload + padding);
```

Listing 2. Validation code to remediate Heartbleed; follows line 2569 of Listing 1

```
/* 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;
```

Bibliography

[1] CVE-2014-0160
https://cve.mitre.org/cgi-bin/cvename.cgi?name=2014-0160

[2] Attack of the week: OpenSSL Heartbleed
Matthew Green
http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html

[3] Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug
Chris Williams
http://www.theregister.co.uk/2014/04/09/heartbleed_explained

[4] Everything you need to know about the Heartbleed SSL bug
Troy Hunt
http://www.troyhunt.com/2014/04/everything-you-need-to-know-about.html

[5] How Heartbleed Works: The Code Behind the Internet's Security Nightmare
Eric Limer
http://gizmodo.com/how-heartbleed-works-the-code-behind-the-internets-se-1561341209

[6] Diagnosis of the OpenSSL Heartbleed Bug
Sean Cassidy
http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html

[7] On Detecting Heartbleed with Static Analysis
Andy Chou, Coverity
http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html

[8] OpenSSL
https://www.openssl.org

[9] The Transportation Layer Security (TLS) Protocol Version 1.2 - IETF RFC 5246
http://tools.ietf.org/html/rfc5246

[10] Datagram Transport Layer Security Version 1.2 - IETF RFC 6347
http://tools.ietf.org/html/rfc6347

[11] TLS and DTLS Heartbeat Extension - IETF RFC 6520
http://tools.ietf.org/html/rfc6520

[12] CWE-130: Improper Handling of Length Parameter Inconsistency
https://cwe.mitre.org/data/definitions/130.html

[13] OpenSSL patch fixing the Heartbleed vulnerability
http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db9023b881d7cd9f379b0c154650d6c108e9a3;hp=0d7717fc9c83dafab8153cbd5e2180e6e04cc802

[14] CWE-125: Out-of-bounds Read
https://cwe.mitre.org/data/definitions/125.html

[15] Valgrind
http://valgrind.org

[16] gmane.os.openbsd.misc post regarding OpenSSL preventing run-time defenses
Theo de Raadt
http://article.gmane.org/gmane.os.openbsd.misc/211963