

**Tuesday, April 7. 2015**

## How Heartbleed could've been found

***tl;dr** With a reasonably simple fuzzing setup I was able to rediscover the Heartbleed bug. This uses state-of-the-art fuzzing and memory protection technology (american fuzzy lop and Address Sanitizer), but it doesn't require any prior knowledge about specifics of the Heartbleed bug or the TLS Heartbeat extension. We can learn from this to find similar bugs in the future.*

Exactly one year ago a bug in the OpenSSL library became public that is one of the most well-known security bug of all time: [Heartbleed](#). It is a bug in the code of a TLS extension that up until then was rarely known by anybody. A read buffer overflow allowed an attacker to extract parts of the memory of every server using OpenSSL.

## Can we find Heartbleed with fuzzing?

Heartbleed was introduced in OpenSSL 1.0.1, which was released in March 2012, two years earlier. Many people wondered how it could've been hidden there for so long. [David A. Wheeler wrote an essay](#) discussing how fuzzing and memory protection technologies could've detected Heartbleed. It covers many aspects in detail, but in the end he only offers speculation on whether or not fuzzing would have found Heartbleed. So I wanted to try it out

Of course it is easy to find a bug if you know what you're looking for . As best as reasonably possible I tried not to use any specific information I had about Heartbleed. I created a setup that's reasonably simple and similar to what someone would also try it without knowing anything about the specifics of Heartbleed.

Heartbleed is a read buffer overflow. What that means is that an application is reading outside the boundaries of a buffer. For example, imagine an application has a space in memory that's 10 bytes long. If the software tries to read 20 bytes from that buffer, you have a read buffer overflow. It will read whatever is in the memory located after the 10 bytes. These bugs are fairly common and the basic concept of exploiting buffer overflows is pretty old. Just to give you an idea how old: Recently the [Chaos Computer Club celebrated the 30th anniversary of a hack of the German BtX-System](#), an early online service. They used a buffer overflow that was in many aspects very similar to the Heartbleed bug. (It is actually disputed if this is really what happened, but it seems reasonably plausible to me.)

Fuzzing is a widely used strategy to find security issues and bugs in software. The basic idea is simple: Give the software lots of inputs with small errors and see what happens. If the software crashes you likely found a bug.

When buffer overflows happen an application doesn't always crash. Often it will just read (or write if it is a write overflow) to the memory that happens to be there. Whether it crashes depends on a lot of circumstances. Most of the time read overflows won't crash your application. That's also the case with Heartbleed. There are a couple of technologies that improve the detection of memory access errors like buffer overflows. An old and well-known one is the debugging tool Valgrind. However Valgrind slows down applications a lot (around 20 times slower), so it is not really well suited for fuzzing, where you want to run an application millions of times on different inputs.

## Address Sanitizer finds more bug

A better tool for our purpose is [Address Sanitizer](#). David A. Wheeler calls it “nothing short of amazing”, and I want to reiterate that. I think it should be a tool that every C/C++ software developer should know and should use for testing.

Address Sanitizer is part of the C compiler and has been included into the two most common compilers in the free software world, gcc and llvm. To use Address Sanitizer one has to recompile the software with the command line parameter `-fsanitize=address`. It slows down applications, but only by a relatively small amount. [According to their own numbers](#), an application using Address Sanitizer is around 1.8 times slower. This makes it feasible for fuzzing tasks.

For the fuzzing itself a tool that recently gained a lot of popularity is [american fuzzy lop \(afl\)](#). This was developed by Michal Zalewski from the Google security team, who is also known by his nick name lcamtuf. As far as I'm aware the approach of afl is unique. It adds instructions to an application during the compilation that allow the fuzzer to detect new code paths while running the fuzzing tasks. If a new interesting code path is found then the sample that created this code path is used as the starting point for further fuzzing.

Currently `all` only uses file inputs and cannot directly fuzz network input. OpenSSL has a command line tool that allows all kinds of file inputs, so you can use it for example to fuzz the certificate parser . But this approach does not allow us to directly fuzz the TLS connection, because that only happens on the network layer . By fuzzing various file inputs I recently found two issues in OpenSSL, but both had been found by Brian Carpenter before, who at the same time was also fuzzing OpenSSL.

## Let OpenSSL talk to itself

So to fuzz the TLS network connection I had to create a workaround. I wrote a small [application that creates two instances of OpenSSL that talk to each other](#). This application doesn't do any real networking, it is just passing buffers back and forth and thus doing a TLS handshake between a server and a client. Each message packet is written down to a file. It will result in six files, but the last two are just empty, because at that point the handshake is finished and no more data is transmitted. So we have four files that contain actual data from a TLS handshake. If you want to dig into this, a [good description of a TLS handshake is provided by the developers of OCaml-TLS and MirageOS](#).

Then I added the possibility of switching out parts of the handshake messages by files I pass on the command line. By calling my test application selfts with a number and a filename a handshake message gets replaced by this file. So to test just the first part of the server handshake I'd call the test application, take the output file packet-1 and pass it back again to the application by running selfts 1 packet-1. Now we have all the pieces we need to use american fuzzy lop and fuzz the TLS handshake.

I compiled OpenSSL 1.0.1f, the last version that was vulnerable to Heartbleed, with american fuzzy lop. This can be done by calling `./config` and then replacing gcc in the Makefile with `afl-gcc`. Also we want to use Address Sanitizer, to do so we have to set the environment variable `AFL_USE_ASAN` to 1.

There are some issues when using Address Sanitizer with american fuzzy lop. Address Sanitizer needs a lot of virtual memory (many Terabytes). American fuzzy lop limits the amount of memory an application may use. It is not trivially possible to only limit the real amount of memory an application uses and not the virtual amount, therefore american fuzzy lop cannot handle this flawlessly. Different solutions for this problem have been proposed and are currently developed. I usually go with the simplest solution: I just disable the memory limit of afl (parameter -m -1). This poses a small risk: A fuzzed input may lead an application to a state where it will use all available memory and thereby will cause other applications on the same system to malfunction. Based on my experience this is very rare, so I usually just ignore that potential problem.

After having compiled OpenSSL 1.0.1f we have two files libssl.a and libcrypto.a. These are static versions of OpenSSL and we will use them for our test application. We now also use the afl-gcc to compile our test application:

```
AFL USE ASAN=1 afl-gcc selftls.c -o selftls libssl.a libcrypto.a -ldl
```

Now we run the application. It needs a dummy certificate. I have put one in the repo. To make things faster I'm using a 512 bit RSA key. This is completely insecure, but as we don't want any security here – we just want to find bugs – this is fine, because a smaller key makes things faster. However if you want to try fuzzing the latest OpenSSL development code you need to create a larger key, because it'll refuse to accept such small keys.

The application will give us six packet files, however the last two will be empty. We only want to fuzz the very first step of the handshake, so we're interested in the first packet. We will create an input directory for american fuzzy lop called in and place packet-1 in it. Then we can run our fuzzing job:



## Quicksearch

## About me

Informationen über meine Arbeit als freier Journalist finden Sie [hier](#).

**Hanno Böck**  
mail: [hanno@hboeck.de](mailto:hanno@hboeck.de)  
jabber: [hanno@hboeck.de](mailto:hanno@hboeck.de)  
[GPG/OpenPGP key](#)

[Hanno on Google+](#)  
[Hanno on Twitter](#)  
[Hanno on identi.ca](#)

Impressum



## Tags

[asia](#) [asia2013](#) [atomkraft](#) [baku](#) [berlin](#)  
[bibi](#) [bundesrat](#) [canc](#) [cnc](#) [copyright](#) [china](#) [csc](#)  
[copyright](#) [creativecommons](#)  
[cryptograph](#) [datenschutz](#)  
[demonstration](#) [demosene](#) [encryption](#) [english](#)  
[ertrinken](#) [essen](#) [finnen](#) [film](#) [freeculture](#)  
[freesoftware](#) [games](#) [gates](#) [german](#) [gentoo](#)  
[gibt](#) [gib](#) [hardware](#) [hies](#) [hieser](#) [karslsruhe](#)  
[karlsruhe](#) [kino](#) [kino](#) [klima](#) [klimaschutz](#)  
[klimawandel](#) [kochen](#) [kohlmarkt](#) [kroatien](#) [linux](#) [lug](#)  
[mord](#) [mordmord](#) [murdard](#) [murdard](#) [notebook](#) [open](#)  
[openstreetmap](#) [open](#) [open](#) [open](#) [open](#) [open](#) [open](#)  
[privacy](#) [religion](#) [ruhr](#) [russland](#)  
[security](#) [sen](#) [sen](#) [sen](#) [ssl](#) [stuttgart](#)  
[supermarkt](#) [tas](#) [tas](#) [travel](#) [trip2011](#)  
[umwelt](#) [umweltschutz](#) [video](#) [vins](#)  
[vorratsdatenschutz](#) [vulnerability](#) [web](#)  
[websecurity](#) [win](#) [win](#) [win](#) [win](#) [ökologie](#)  
[überwachung](#)

## Anzeigen




## Events

## My pages

[Picture gallery](#)

## Feeds

 [RSS 2.0 feed](#)

 [ATOM 1.0 feed](#)

```
afl-fuzz -i in -o out -m -1 -t 5000 ./selftls 1 @@
```

american fuzzy lop 1.57b (selfs)			
process timing		overall results	
run time : 1 days, 11 hrs, 14 min, 16 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 6 min, 19 sec		total paths : 631	
last uniq crash : 0 days, 20 hrs, 53 min, 47 sec		uniq crashes : 4	
last uniq hang : 0 days, 1 hrs, 35 min, 13 sec		uniq hangs : 38	
cycle progress		map coverage	
now processing : 616 (97.62%)		map density : 8353 (12.75%)	
paths timed out : 0 (0.00%)		count coverage : 1.85 bits/tuple	
stage progress		findings in depth	
now trying : interest 16/8		favored paths : 156 (24.72%)	
stage execs : 3738/8176 (45.72%)		new edges on : 228 (36.13%)	
total execs : 8.43M		total crashes : 487 (4 unique)	
exec speed : 53.44/sec (slow!)		total hangs : 159 (38 unique)	
fuzzing strategy yields		path geometry	
bit flips : 158/373k, 28/372k, 28/372k		levels : 7	
byte flips : 0/46.6k, 1/32.6k, 2/36.7k		pending : 427	
arithmetics : 120/1.64M, 29/1.34M, 6/385k		pend fav : 7	
known ints : 12/148k, 44/1.00M, 52/1.79M		own finds : 630	
dictionary : 0/0, 0/0, 13/399k		imported : n/a	
havoc : 108/446k, 0/0		variable : 371	
trim : 1.99%/20.2k, 35.69%			
[cpu: 52%]			

We pass the input and output directory , disable the memory limit and increase the timeout value, because TLS handshakes are slower than common fuzzing tasks. On my test machine around 6 hours later afl found the first crash. Now we can manually pass our output to the test application and will get a stack trace by Address Sanitizer:

```
==2268==ERROR: AddressSanitizer: heap-buf-fer-overflow on address 0x629000013748 at pc 0x7f228f5f0cfa bp 0x7f ffe8dbd590 sp 0x7ffe8dbcd38
READ of size 32768 at 0x629000013748 thread T0
#0 0x7f228f5f0cf9 (/usr/lib/gcc/x86_64-pc-linux-gnu/4.9.2/libasan.so.1+0x2fcf9)
#1 0x43d075 in memcpy /usr/include/bits/string3.h:51
#2 0x43d075 in tls1_process_heartbeat /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/ssl/tl1_lib.c:2586
#3 0x50e498 in ssl3_read_bytes /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/ssl/s3_pkt.c:1092
#4 0x51895c in ssl3_get_message /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/ssl/s3_both.c:457
#5 0x4ad90b in ssl3_get_client_hello /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/ssl/s3_srvr.c:941
#6 0x4c831a in ssl3_accept /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/ssl/s3_srvr.c:357
#7 0x412431 in main /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/selfs.c:85
#8 0x7f228f03ff9f in __libc_start_main (/lib64/libc.so.6+0x1ff9f)
#9 0x4252a1 (/data/openssl/openssl-handshake/openssl-1.0.1f-nobreakrng-afl-asan-fuzz/selfs+0x4252a1)
```

0x629000013748 is located 0 bytes to the right of 17736-byte region [0x62900000f200,0x629000013748)

allocated by thread T0 here:

```
#0 0x7f228f6186f7 in malloc (/usr/lib/gcc/x86_64-pc-linux-gnu/4.9.2/libasan.so.1+0x576f7)
#1 0x57f026 in CRYPTO_malloc /home/hanno/code/openssl-fuzz/tests/openssl-1.0.1f/crypto/mem.c:308
```

We can see here that the crash is a heap buf-fer overflow doing an invalid read access of a round 32 Kilobytes in the function `tls1_process_heartbeat()`. It is the Heartbleed bug. We found it.

I want to mention a couple of things that I found out while trying this. I did some things that I thought were necessary , but later it turned out that they weren't. After Heartbleed broke the news a number of reports stated that Heartbleed was partly the fault of OpenSSL's memory management. A [mail by Theo De Raadt claiming that OpenSSL has "exploit mitigation countermeasures"](#) was widely quoted. I was aware of that, so I first tried to compile OpenSSL without its own memory management. That can be done by calling `./config` with the option `no-buf-freelist`.

But it turns out although OpenSSL uses its own memory management that doesn't defeat Address Sanitizer . I could replicate my fuzzing finding with OpenSSL compiled with its default options. Although it does its own allocation management, it will still do a call to the system's normal `malloc()` function for every new memory allocation. A [blog post by Chris Rohlf digs into the details of the OpenSSL memory allocator](#).

#### Breaking random numbers for deterministic behaviour

When fuzzing the TLS handshake american fuzzy lop will report a red number counting variable runs of the application. The reason for that is that a TLS handshake uses random numbers to create the master secret that's later used to derive cryptographic keys. Also the RSA functions will use random numbers. I wrote a [patch to OpenSSL to deliberately break the random number generator](#) and let it only output ones (it didn't work with zeros, because OpenSSL will wait for non-zero random numbers in the RSA function).

During my tests this had no noticeable impact on the time it took afl to find Heartbleed. Still I think it is a good idea to remove nondeterministic behavior when fuzzing cryptographic applications. Later in the handshake there are also timestamps used, this can be circumvented with [libfaketime](#), but for the initial handshake processing that I fuzzed to find Heartbleed that doesn't matter .

#### Conclusion

You may ask now what the point of all this is. Of course we already know where Heartbleed is, it has been patched, fixes have been deployed and it is mostly history . It's been analyzed thoroughly .

The question has been asked if Heartbleed could've been found by fuzzing. I'm confident to say the answer is yes. One thing I should mention here however: American fuzzy lop was already available back then, but it was barely known. It only received major attention later in 2014, after [Michal Zalewski used it to find two variants of the Shellshock bug](#) . Earlier versions of afl were much less handy to use, e. g. they didn't have 64 bit support out of the box. I remember that I failed to use an earlier version of afl with Address Sanitizer , it was only possible after a couple of issues were fixed. A lot of other things have been improved in afl, so at the time Heartbleed was found american fuzzy lop probably wasn't in a state that would've allowed to find it in an easy , straightforward way .

I think the takeaway message is this: We have powerful tools freely available that are capable of finding bugs like Heartbleed. We should use them and look for the other Heartbleeds that are still lingering in our software. Take a look at the [Fuzzing Project](#) if you're interested in further fuzzing work. There are beginner tutorials that I wrote with the idea in mind to show people that fuzzing is an easy way to find bugs and improve software quality .

I already used my sample application to fuzz the latest OpenSSL code. Nothing was found yet, but of course this could be further tweaked by trying different protocol versions, extensions and other variations in the handshake.

I also wrote a [German article about this finding for the IT news webpage Golem.de](#) .

Update:

Creative Commons



Unless noted otherwise, all content is CC Zero / public domain

I want to point out some feedback I got that I think is noteworthy .

On [Twitter it was mentioned that Codenomicon actually found Heartbleed via fuzzing](#) . There's a [Youtube video from Codenomicon's Antti Kariäläinen explaining the details](#) . However the way they did this was quite different, they built a protocol specific fuzzer . The remarkable feature of all is that it is very powerful without knowing anything specific about the used protocol. Also it should be noted that Heartbleed was found twice, the first one was Neel Mehta from Google.

Kostya Serebryany mailed me that he was [able to replicate my findings with his own fuzzer which is part of LL-VM](#), and it was even faster .

In the comments Michele Spagnuolo mentions that by compiling OpenSSL with `-DOPENSSL_TLS_SECURITY_LEVEL=0` one can use very short and insecure RSA keys even in the latest version. Of course this shouldn't be done in production, but it is helpful for fuzzing and other testing efforts.

Posted by [Hanno Böck](#) in [Code](#), [Cryptography](#), [English](#), [Gentoo](#), [Linux](#), [Security](#) at [15:23](#) | [Comments \(3\)](#) | [Trackbacks \(4\)](#)

Defined tags for this entry: [addresssanitizer](#), [all](#), [americanfuzzylop](#), [bufferoverflow](#), [fuzzing](#), [heartbleed](#), [openssl](#)

Related entries by tags:

[Safer use of C code - running Gentoo with Address Sanitizer](#)  
[Dancing protocols. POODLEs and other tales from TLS](#)  
[How to stop Bleeding Hearts and Shocking Shells](#)  
[LibreSSL on Gentoo](#)  
[Fuzzing is easy](#)

2

## Trackbacks

[Trackback specific URI for this entry](#)

### [PingBack](#)

**Weblog:** [www.btk-clan.ch](#)  
**Tracked:** Apr 08, 07:57

### [PingBack](#)

**Weblog:** [snippets.mela.de](#)  
**Tracked:** Apr 09, 06:32

### [PingBack](#)

**Weblog:** [info.ssl.com](#)  
**Tracked:** Apr 30, 02:51

### [Denial of Service in Dovecot and unexpected crashes in OpenSSL \(TFPA 008/2015\)](#)

A while ago I did a little experiment trying to fuzz the OpenSSL handshake with the intent to test whether Heartbleed could've been found with fuzzing. At some point while developing the sample code I discovered that american fuzzy lop would find a lot of

**Weblog:** [The Fuzzing Project](#)  
**Tracked:** May 18, 21:15

## Comments

Display comments as ([Linear](#) | [Threaded](#))

The main thing about HeartBleed was not technical, there were N ways it could have been found.

No, the main thing is that nobody had or took the time to even look such bugs.

As I wrote in ACM Queue: "Quality Software costs money, Heartbleed is free"

<https://queue.acm.org/detail.cfm?id=2636165>

Poul-Henning

[#1](#) Poul-Henning Kamp on 2015-04-07 21:43 ([Reply](#))

You can make OpenSSL accept any key length by compiling it with `-DOPENSSL_TLS_SECURITY_LEVEL=0`. Of course this is not safe in production, but it is fine to fuzz it with very short keys.

[#2](#) Michele Spagnuolo ([Homepage](#)) on 2015-04-08 12:01 ([Reply](#))

As mentioned before on Twitter (<https://twitter.com/jurajsomorovsky/status/590867082612502528>), I try to test Heartbleed on OpenSSL with Address Sanitizer. My problem is that the server crashes only when the heartbeat length value > 17709. Otherwise, the server runs further as it would not have been compiled with Address Sanitizer.

As you mentioned, I tried to compile OpenSSL with `no-buf-free-lists`:

```
.config no-buf-free-lists -fsanitize=address
make depend
make
But this does not change the server behavior.
```

Btw., when I set the length value to 17710, the server crashes and I get the following output:  
0x629000022749 is located 0 bytes to the right of 17736-byte region [0x62900001e200,0x629000022748]

When I increment the length value to 17711, I get the following message:  
0x629000018749 is located 1 bytes to the right of 17736-byte region [0x629000014200,0x629000018748]  
etc.

It seems to be some fixed 17736-byte region, which must be overread, otherwise the server does not crash.

Do you mean is this something connected with Address Sanitizer properties or with the OpenSSL implementation?

I would like to do some fuzzing and to configure Address Sanitizer to be more "sensitive" (if that's possible)...

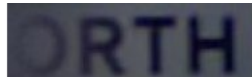
Thanks  
Juraj

[#3](#) Juraj on 2015-04-22 17:47 ([Reply](#))

Add Comment

---

Name	<input type="text"/>
Email	<input type="text"/>
Homepage	<input type="text"/>
In reply to	<span>[ Top level ]</span> ▼
Comment	<div></div>

[Privacy & Terms](#)

E-Mail addresses will not be displayed and will only be used for E-Mail notifications.

☐ Remember Information?

☐ Subscribe to this entry