Security

Avoiding Cyberspace Catastrophes through Smarter Testing

Apostol Vassilev
National Institute of Standards and Technology

Christopher Celi
Rensselaer Polytechnic Institute

The Heartbleed bug highlighted a critical problem in the software industry: inadequately tested software results in serious security vulnerabilities. Available testing technologies, combined with emerging standards, can help tech companies meet increasing consumer demand for greater Internet security.

Social networking, the Internet of Things, advances in medical technology such as the brain-controlled, electrode-enabled exoskeleton that allowed a young paraplegic to deliver the 2014 FIFA World Cup kickoff in Brazil—more and more, IT is entwined in the fabric of our lives. But just as boundaries between our real and virtual selves blur, security breaches in IT can now have very real—even catastrophic—consequences.

The question arises: Is software, which constitutes the largest part of IT, designed for optimal security, implemented correctly, and thoroughly tested? As reported data breaches in 2013 alone accounted for over 800 million online records being compromised,[1] the answer too often is apparently "no"—a predicament that has led *The Economist* to declare a market failure in cybersecurity, citing as a main reason the way computer code is produced. Commercial software is often shipped as quickly as possible, aiming to beat competitors' offerings to market and hoping to patch defects as they are discovered in the field. This makes it hard for users to assess cybersecurity risks and develop mitigations based on software security controls.

General Solutions

One strategy that can help correct the problem is developing standards to encourage improved security. In February, the National Institute of Standards and Technology (NIST) published the Framework for Improving Critical Infrastructure Cybersecurity (www.nist.gov/cyberframework), a set of voluntary guidelines designed for companies in critical infrastructure sectors such as energy and transportation. These guidelines allow even small companies to apply risk management principles to better utilize security technologies for their software. As organizations adopt the framework, a stronger focus on the robustness in software security controls should create incentives for software companies to improve in this area.

Societal and market pressures can also play a constructive role in improving the status quo for security. A good example here is how safety has evolved in the automotive industry.[1] Volvo was perhaps the first auto manufacturer to market its safety rating and turn it into a competitive advantage,[2] and today most car companies present safety as an important product feature just as consumers prefer by significant margins an option that has better government safety ratings.[3]

The software industry could likewise profit by meeting consumer expectations for data protection and cybersecurity overall.

A "Model" for Cybersecurity Catastrophe: Heartbleed

While many cybersecurity threats have surfaced in recent years,[1] perhaps none is more widely known than the Heartbleed bug in OpenSSL—by far the most common implementation of the Secure Sockets Layer and Transport Layer Security (SSL/TLS) protocols according to which sensitive data sent across the Internet is routinely encrypted (http://computer.howstuffworks.com/encryption4.htm). In theory, these protocols provide effective Internet security. However, in practice, specific SSL/TLS implementations are susceptible to common programming errors, which can lead to disastrous security holes and potentially compromise data confidentiality and integrity on a very large scale. For example, *Ars Technica* reported 615,268 vulnerable servers on the Internet when Heartbleed was first discovered in April 2014, with 318,239 still vulnerable a month later,[4] a situation security expert Bruce Schneier called "catastrophic."[5]

The potential damage that could have resulted from disclosure of sensitive data caused software patching on a massive scale. Moreover, the time and resources spent upgrading systems so they could use a patched version of OpenSSL created substantial actual costs for Internet service providers and certificate authorities alike.

Dissecting the Error

The bug, first reported widely by the software security testing organization Codenomicon, allows an attacker to overread a buffer, grabbing up to 64 KB of server memory at a time—data that includes anything from login IDs and passwords to credit card numbers. Undetectable, simple to perform, and easily repeatable, the attack has been possible since December 2011 (http://heartbleed.com).

The bug was introduced by a faulty implementation of the TLS heartbeat extension—specifically involving a client issuing heartbeat requests and a server's corresponding heartbeat responses—that allows a secure connection to stay alive without the session having to be renegotiated (http://tools.ietf.org/html/rfc6520). Because a simple bounds check on a size parameter sent with a heartbeat request was omitted in the feature's implementation in the OpenSSL source code, more than half a million websites were exposed to the Heartbleed attack.[6]

Buffer overreads can be particularly dangerous in server-hosted applications because servers typically host multiple applications running under the same system account for multiple remote users. Thus, data leaked by one application responding to a request could contain data belonging to another user request in the same application or to a separate application thought to be completely secure. Heartbleed was indiscriminate in the data leaked: it allowed the next 64 KB of data in a server's memory related to processing an incoming heartbeat request to be returned.

Preventing the Next Heartbleed

How can an organization, such as the community responsible for developing and maintaining OpenSSL, prevent bugs like Heartbleed? One obvious step is to adopt smarter testing standards.

Positive and negative testing

For testing software, the two most common approaches involve positive and negative testing.[7] A positive test focuses on ensuring that a feature performs correctly with expected input. A negative test, by contrast, serves to ensure that a feature doesn't behave badly with invalid data as input. A failed negative test result could range from a buffer overread or overwrite, with consequent data loss, to a program crash, with a potential privilege elevation being granted. The goal in negative testing is for a system to gracefully handle unexpected input and continue running without data being lost or leaked.

All systems undergo at least some positive testing to make sure the product performs correctly under normal usage scenarios. However, judging from reported cybersecurity breaches, many systems appear to be subjected, at best, to lackluster negative testing, leaving security holes in the implementation.

To understand negative testing, consider that while a program typically has only a limited number of *acceptable* input values, the number of *unexpected* input values is essentially infinite. For example, if a simple calculator program expects two numbers and an operator (+, -, *, or /), the expected input is clearly limited. But how would the program handle a random letter string entered as one of the numbers or an "x" to indicate multiplication instead of "*"? While simple cases like these can be flushed out easily with negative testing, how can a test cover the full range of potential unexpected input?

Test tools called fuzzers address this problem (www.cgisecurity.com/questions/securityfuzzer.shtml). A fuzzer takes an expected input, mutates it randomly, and sends the mutation to the system being tested to cause the system to misbehave. Fuzzers allow efficient automatic software testing by continually generating new inputs based on the original. If it generates enough random inputs to span the range of possible input parameters, a fuzzer performs negative testing. In practice, however, because the large input parameter range required for fuzzing results in high computational costs and takes time, organizations tend to perform such testing on only a minimal number of system components.

Static and dynamic analysis

Software testing technologies are also distinguishable by the type of analysis they perform on target applications or systems. Some test suites evaluate program source code using static analysis tools—those that work without executing a target program—to detect blocks of potentially dangerous code. For example, static analysis could have prevented the "goto fail" bug, another high-profile software vulnerability reported by Apple in the TLS implementation for its mobile devices (http://security.coverity.com/blog/2014/Feb/a-quick-post-on-apple-security-55471-aka-goto-fail.html). This bug caused Apple's browsers to recognize invalid certificates and establish connections that could potentially link unsuspecting users to malicious servers. The specific coding error rendered a section of code unreachable, a condition that many static analysis tools can detect and report.

However, particularly with complex code, static analyzers often have fairly limited functionality and often miss implementation errors with severe security consequences: the more complex the source code, the less amenable it is to machine analysis. In addition, some coding errors can only be detected at runtime. In C and C++, for example, it isn't necessarily an error to read from or write to an address likely to be determined by specific variables at runtime, so static analyzers throw no warnings when encountering such code. As a result, these tools aren't generally able to detect Heartbleed-like errors.

In cases like these, dynamic analysis tools—those that require a program to run to detect vulnerabilities—are more likely to succeed. Dynamic analysis tools use various methods, such as guard pages, to detect and prevent buffer overreads—the problem with Heartbleed—and overwrites at runtime. Using the automotive analogy, while static analysis is like reviewing blueprints and concept art for a car, dynamic analysis is akin to taking a car on a test drive (http://istqbexamcertification.com/what-is-dynamic-analysis-tools-in-software-testing). While much more time consuming, dynamic analysis detects potential problems far more accurately than simple static analysis can.

Some open source products available for dynamic analysis include Valgrind (http://valgrind.org) and AddressSanitizer (http://code.google.com/p/address-sanitizer).

Valgrind. Valgrind dynamically analyzes compiled programs, targeting memory errors and leaks. Memcheck, probably Valgrind's most commonly used plug-in, performs a check to verify that memory has been allocated properly and initialized correctly for every memory access in an executable. Memcheck can detect buffer overreads and buffer overwrites, in addition to memory errors such as leaks. Though Valgrind leads to a performance slowdown of as much as 40 times when it's run during test phases (http://valgrind.org/docs/manual/manual-core.html), once a system is debugged and ready for use, the software runs with normal performance characteristics.

To determine Valgrind's effectiveness, we created a test environment with instances of Valgrind and OpenSSL to see if it could detect Heartbleed. Versions of OpenSSL prior to version 1.0.1g are vulnerable, so our test machine ran OpenSSL 1.0.1e, released in February 2013 (http://openssl.org), with the latest version of Valgrind, 3.10.0.

Our tests showed that running OpenSSL under Valgrind and establishing a server locally using the native server command *via valgrind openssl s_server [flags]* caused an error to be raised when the server processed a malicious heartbeat request. (As expected, the server did not terminate but rather continued executing and even processing subsequent heartbeat requests.) The following is a sample from the error trace:

```
==18005== Invalid read of size 8

==18005==    at 0x4C2F79E: memcpy@@GLIBC_2.14 (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)

==18005==    by 0x4E6B196: tls1_process_heartbeat (in /lib/libssl.so.1.0.0)

==18005==    by 0x4E62F10: ssl3_read_bytes (in /lib/libssl.so.1.0.0)
```

Valgrind detected an invalid read during the tls1_process_heartbeat method, the method responsible for parsing a heartbeat request and returning data to the client. This method contains the programming error responsible for Heartbleed. The error trace above would allow a programmer to investigate the input that led Valgrind to the error, identify the code responsible for the buffer overread, and fix the tls1_process_heartbeat method.

AddressSanitizer. AddressSanitizer (ASan) is a readily available compiler option for GCC versions 4.8 and later (www.gnu.org/software/gcc) and for LLVM/Clang (http://clang.llvm.org) that is highly useful in detecting memory errors. ASan is a compiler flag and thus requires a full system compilation to be used. ASan can cause memory overhead of between 2 and 4 times and a 73 percent performance overhead;[6] as with Valgrind, this slowdown occurs only during testing—and is a negligible cost when software security is at stake.

Adding the ASan flag -fsanitize=address while running the configure file for OpenSSL resulted in an OpenSSL executable, with the same runtime checking capabilities as ASan built-in. ASan, when compiled with an executable, allocates a large section of virtual memory and initializes it as one particular value. Each memory access is then compared with the original value to ensure that memory has been allocated properly before use. We do note that ASan should only be used for testing purposes, not on a production server, because, by design, ASan terminates the compiled program upon detecting a memory error.

We tested ASan's capabilities by sending different heartbeat requests to a server. We ran the same server command as before, openssl s_server [flags], this time on OpenSSL compiled with the ASan flag. When a normal heartbeat request is sent to the server, no unusual behavior occurs, and the server maintains function, as expected. But when a malicious heartbeat request is sent in an attempt to get the server to dump 64 KB of information to the client, the server terminated and the following ASan error trace was generated:

```
==3255== ERROR: AddressSanitizer: unknown-crash on address 0x608200016a0b at pc 0x7fa3eadd43f7 bp
0x7fffe7740f10 sp 0x7fffe77406d0

READ of size 65535 at 0x608200016a0b thread T0

  #0 0x7fa3eadd43f6 (/usr/lib/x86_64-linux-gnu/libasan.so.0.0.0+0xe3f6)

  #1 0x7fa3eab339ca (/lib/libssl.so.1.0.0+0x7d9ca)

...

==3255== ABORTING
```

The trace shows an error occurring during "READ of size 65535", which is 64 KB of information. Note, however, that the tls1_process_heartbeat method isn't explicitly mentioned in the trace. Still, the computational context at the time of aborting, including the input, can be used to identify an error reported by ASan.

To ensure that our testing didn't simply find the bug because we know it exists, we included a fuzzer in our test client. This allowed us to specify which protocol parts remained static (such as metadata and the request type) and which were dynamically generated while maintaining a problem-agnostic testing approach. In particular, our setup instrumented the heartbeat request call to incorporate random data in the payload specification sent to the server, thereby allowing us to run the fuzzed tests automatically. Our results yielded errors that were detected for random values forming invalid heartbeat requests. Normal requests yielded no memory errors.

Looking to the Future

Dynamic analysis tools can be combined with fuzzers to substantially increase the scope and depth of software testing, not just for protocol implementations but also for other large software systems. Fuzzers play a key role in improving test suites for software by helping to tackle the infinite input range involved in negative testing. Most fuzzers also allow programmers to select which parts of an input are fuzzed and which parts remain static, an important feature when testing protocols where certain parts of a composite message are expected to contain metadata specifying how the system should treat the rest of the message.

In fact, using fuzzers in conjunction with dynamic analysis tools is included in emerging software development standards such as ISO/IEC 27034 (http://www.iso27001security.com/html/27034.html), a set of guidelines addressing secure software development from the outset, as well as in Microsoft's Security Development Lifecycle (www.microsoft.com/security/sdl/default.aspx). But so far these standards haven't achieved wide industry adoption. Heartbleed might have pushed the software industry to recognize the value of such comprehensive testing technologies and methodologies.

No organization should depend on one single testing tool; errors can still be falsely identified or missed entirely. Incorporating multiple tools increases detection dramatically. Still, developing a complete suite of reliable test tools able to find every software security problem requires further investigation.

As a practical matter though, organizations have a plethora of open source tools available such as Valgrind and AddressSanitizer, in addition to commercial tools such as IBM's Rational Purify (www-03.ibm.com/software/products/en/ratpurwin), that can be used to better test software. There are also many open source fuzzers such as SPIKE (http://resources.infosecinstitute.com/intro-to-fuzzing) and Simple Fuzzer (http://aconole.brad-x.com/programs/sfuzz.html) as well as commercially available products such as Codenomicon's test suite (www.codenomicon.com/products/protocols.shtml).

Following its mission to promote better software security through voluntary standards and initiatives, NIST sponsored a paper that lists and evaluates various testing tools (NAVSEA*;* http://samate.nist.gov/docs/NAVSEA-Tools-Paper-2009-03-02.pdf), including static and dynamic testing tools as well as fuzzers. While the list is extensive, it's nowhere near complete; useful and excellent tools might well exist that have been developed since the NAVSEA paper's publication.

The main issue lies not in finding tools and assembling a well-endowed testing suite, but in the commitment organizations must make to test and push the bounds of quality assurance and security their software provides. Time and resources will have to be dedicated to thorough testing. While many regard such testing as having little economic benefit, eliminating unpatched security holes reduces the overall cost both for the organization developing the software and for its clients. In fact, NIST's NAVSEA paper established that directing sufficient efforts toward safety and security testing actually reduces total development time because less time is required for fixing issues with the system that arise later.

Heartbleed wasn't as catastrophic a vulnerability as it could have been because it was discovered by responsible people. Unfortunately, today it's difficult to feel confident that a security hole will be reported and not exploited. With the Internet of Things promising to link all sorts of devices and data to the Internet, users will increasingly seek systems that boast more complete security assurance. Standards that help rate systems according to meaningful, easy-to-understand benchmarks similar to those for car safety ratings would help in this regard. In a future where our real and virtual lives increasingly unite, users at all levels will want to feel confident that their sensitive data is protected.

Disclaimer

The identification of any commercial product or trade name does not imply endorsement or recommendation by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

References

1. "Special Report: Cyber-Security," *The Economist*, 12 July 2014; www.economist.com/sites/default/files/20140712_cyber-security.pdf.

2. P. George, "Volvo Gave Away Their Most Important Invention to Save Lives," *Jalopnik*, 8 Aug. 2013; http://jalopnik.com/volvo-gave-away-their-most-important-invention-to-save-1069825878.

3. A. Linde, "How Are Car Safety Ratings Affecting Sales Figures," blog, 31 Aug. 2013; www.drivingsales.com/blogs/new-marketing-auto-business/2013/08/31/how-car-safety-ratings-affecting-sales-figures.

4. D. Goodin, "Four Weeks on, Huge Swaths of the Internet Remain Vulnerable to Heartbleed," *Ars Technica,* 8 May 2014; http://arstechnica.com/security/2014/05/four-weeks-on-huge-swaths-of-the-internet-remain-vulnerable-to-heartbleed.

5. B. Schneier, "Heartbleed,"blog, 9 Apr. 2104; https://www.schneier.com/blog/archives/2014/04/heartbleed.html

6. Mutton, "Half a Million Widely Trusted Websites Vulnerable to Heartbleed Bug," *Netcraft,* 8 Apr. 2014; http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html.

7. D. Wheeler, "How to Prevent the Next Heartbleed," 9 Sept. 2013; www.dwheeler.com/essays/heartbleed.html.

Apostol Vassilev *is a cybersecurity expert in the Computer Security Division of the National Institute of Standards and Technology (NIST). His research interests include cybersecurity standards and cryptography. Vassilev received a PhD in mathematics from Texas A&M University. Contact him at apostol.vassilev@nist.gov.*

Christopher Celi *is an undergraduate at Rensselaer Polytechnic Institute studying computer science and mathematics. He worked over the summer in the Computer Security Division at NIST as part of its Security Testing, Validation, and Measurement Group. Contact him at celic@rpi.edu.*

Editor: Jeffrey Voas, National Institute of Standards and Technology; jvoas@ieee.org