Finding More Than One Worm in the Apple

If you see something, say something

Mike Bland

In February Apple revealed and fixed an SSL (Secure Sockets Layer) vulnerability that had gone undiscovered since the release of iOS 6.0 in September 2012. It left users vulnerable to man-in-the-middle attacks thanks to a short circuit in the SSL/TLS (Transport Layer Security) handshake algorithm introduced by the duplication of a `goto` statement. Since the discovery of this very serious bug, many people have written about potential causes. A close inspection of the code, however, reveals not only how a unit test could have been written to catch the bug, but also how to refactor the existing code to make the algorithm testable—as well as more clues to the nature of the error and the environment that produced it.

This article addresses five big questions about the SSL vulnerability: What was the bug (and why was it bad)? How did it happen (and how didn't it)? How could a test have caught it? Why didn't a test catch it? How can we fix the root cause?

WHAT WAS THE BUG (AND WHY WAS IT BAD)?

The Apple SSL vulnerability, formally labeled CVE-2014-1266, was produced by the inclusion of a spurious, unconditional `goto` statement that bypassed the final step in the SSL/TLS handshake algorithm. According to the National Vulnerability Database (<http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266>) and the CVE (Common Vulnerabilities and Exposure) Standard Vulnerability Entry (<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266>), the bug existed in the versions of iOS, OS X, and the Apple TV operating system shown in table 1.

These formal reports describe the bug as follows: “The `SSLVerifySignedServerKeyExchange` function in `libsecurity_ssl/lib/sslKeyExchange.c` in the Secure Transport feature in the Data Security component...does not check the signature in a TLS Server Key Exchange message, which allows man-in-the-middle attackers to spoof SSL servers by (1) using an arbitrary private key for the signing

TABLE 1 Schedule of affected systems and security updates

System	Vulnerable Versions	Vulnerable Since	Fixed Versions	Patch Date
iOS	6.x - 6.1.5	2012-09-19	6.1.6	2014-02-21
	7.x - 7.0.5	2013-09-18	7.0.6 ¹	2014-02-21
OS X	10.9 - 10.9.1	2013-10-22	10.9.2 ²	2014-02-25
Apple TV	6.x - 6.0.1	2012-09-24	6.0.2	2014-02-21

1. iOS 7.0.6 release notes; <http://support.apple.com/kb/HT6147>.

2. OS X 10.9.2 release notes; <http://support.apple.com/kb/HT6114>.

step or (2) omitting the signing step.” This error is visible by searching for the function name within Apple’s published open-source code (http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c) and looking for this series of statements:

```
if ((err = SSLHashSHA1.update(
    &hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```

Those familiar with the C programming language will recognize that the first `goto fail` is bound to the `if` statement immediately preceding it; the second is executed unconditionally. This is because whitespace, used to nest conditional statements for human readability, is ignored by the C compiler; curly braces must enclose all statements bound to an `if` statement when more than one statement applies.

The other `goto fail` statements appearing throughout the algorithm are a common idiom in C for releasing resources when a function has encountered a fatal error prior to completion. In the flawed code, a successful `update()` call will result in an unconditional jump to the end of the function, before the final step of the algorithm; and the return value will indicate the handshake was successful. In essence, the algorithm gets short-circuited.

For users of Apple’s Safari and other Secure Transport-based applications on the affected platforms, “secure” connections were vulnerable to man-in-the-middle attacks, whereby an attacker in a position to relay messages from a client to a “secure” server across the Internet can impersonate the server and intercept all communications after the bogus handshake. (Users of products incorporating their own SSL/TLS implementations, such as Google Chrome and Mozilla Firefox, were not affected.) Though it is unknown whether this vulnerability was ever exploited, it rendered hundreds of millions of devices (and users) vulnerable over the course of 17 months.

Apple was criticized for patching the vulnerability for iOS devices and Apple TV on Friday, February 21, 2014, making knowledge of the vulnerability widespread, but delaying the patch for OS X Mavericks until the following Tuesday. This four-day window left users who weren’t aware of the iOS patch vulnerable to a now very public exploit.

HOW DID IT HAPPEN (AND HOW DIDN’T IT)?

Many have noted apparently missing factors that could have caught the bug. Coding standards—specifically those enforcing the use of indentation and curly braces—combined with automated style-checking tools and code reviews, might have drawn attention to the repeated statement. An automated merge may have produced the offending extra line, and the developer may have lacked sufficient experience to detect it. Had coverage data been collected, it would have highlighted unreachable code. Compiler and static-analysis warnings also could have detected the unreachable code, though false warnings might have drowned out the signal if such tools weren’t already being used regularly.

Others noted that the code appears to lack unit tests, which likely would have caught the bug. While many of the other tools and practices might have been sufficient to prevent this specific vulnerability, a deeper problem, which ultimately produced the repeated `goto` statement, would have been prevented by proper unit-testing discipline.

Some question whether adequate system testing took place, while others argue that because system testing can’t find every bug, this was merely an unfortunate case of one that happened to

slip through. Others claim use of the `goto` statement and/or deliberate sabotage is to blame. None of these claims stands up to scrutiny.

GOTO NOT “CONSIDERED HARMFUL”

Since it's one of the more popular theories, let's dispatch the argument that the use of `goto` is to blame for this vulnerability. Many have referred to the popular notion that `goto` is “considered harmful,” based on Edsger Dijkstra's letter published in the March 1968 *Communications of the ACM*. This is what Dijkstra actually said in “A Case against the GO TO Statement”: “I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs; but whatever clauses are suggested (e.g., abortion clauses) they should satisfy the requirement that a programmer-independent coordinate system can be maintained to describe the process in a helpful and manageable way.”⁹ In other words, “abortion clauses” to release a function's resources may still rely on `goto`, absent other direct language support.

This C language “abortion clause” idiom is legitimate and well understood, and is directly supported by other languages. For example, in C++, automatic destructors implement the RAII (Resource Acquisition Is Initialization) idiom; Java employs `try/catch/finally` blocks (<http://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html>); Go provides the `defer()`, `panic()`, and `recover()` mechanisms (<http://blog.golang.org/defer-panic-and-recover>); and Python has `try/except/finally` blocks (http://docs.python.org/3/reference/compound_stmts.html#try) and the `with` statement, which is used to implement RAII (http://docs.python.org/3/reference/compound_stmts.html#the-with-statement). Absent these mechanisms, in C this remains a legitimate application of the `goto` statement, lest the code become bloated with repetitive statements or the control structure become nested so deeply as to hinder readability and maintainability.

In fact, a misplaced `return` statement could have produced the same effect. Imagine a macro such as the following had been defined:

```
#define ERROR_EXIT {\
    SSLFreeBuffer(&hashCtx);\
    return err; }
```

Then the bug might have appeared in this incarnation:

```
if ((err = SSLHashSHA1.update(
    &hashCtx, &signedParams)) != 0)
    ERROR_EXIT
    ERROR_EXIT
```

Even enforcing the use of curly braces might not have prevented the error, as they could be mismatched:

```
if ((err = SSLHashSHA1.update(
    &hashCtx, &signedParams)) != 0)
{
```

```

    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(
    &hashCtx, &hashOut)) != 0)
    goto fail;
}

```

The blame for this vulnerability does not lie with the `goto` statement. A proper unit test would have caught the error regardless of how it was written.

CODE DUPLICATION

The handshake algorithm in which the extra `goto` statement appears is duplicated six times throughout the code. Figure 1 shows the algorithm containing the repeated `goto fail` line as it appears in the `SSLVerifySignedServerKeyExchange()` function. Figure 2 shows the block immediately preceding this algorithm. *This duplication is the critical factor leading to the manifestation of the vulnerability*, and it can be traced directly to a lack of unit testing discipline—because of the absence of the craftsmanship and design sense that testable code requires. Someone writing code with unit testing in mind would have ensured only one copy of the algorithm existed—not only because it’s theoretically more proper, but because it would have been easier to test.

The coder could not “smell” (<http://blog.codinghorror.com/code-smells/>) the duplicate code as he or she was writing it—or copying it for the second, third, fourth, fifth, or sixth time! This indicates a pattern of poor habits over time, not a single careless mistake. Ultimately, this speaks to a cultural issue, not a moment of individual error.

HOW COULD A TEST HAVE CAUGHT IT?

Landon Fuller published a proof-of-concept unit test implemented in Objective-C,¹⁰ using the Xcode Testing Framework.² Fuller notes that “there’s no reason or excuse for [the

FIGURE 1

The handshake algorithm containing the `goto fail` bug

```

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

```

FIGURE 2

The duplicate handshake algorithm appearing immediately before the buggy block

```

if(isRsa) {
    /* ... */
    if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashMD5.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashMD5.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
        goto fail;
}

```

SSLVerifySignedServerKeyExchange() function] not being fully tested for” all of the potential error conditions. This proof of concept, however, misses the opportunity to look deeper into the code and provide full test coverage of this particularly critical algorithm—so critical that it appears six times in the same file.

Step one in properly testing the algorithm is to extract it into a separate function—which in itself might have prevented the duplicate `goto fail` that caused the bug, since a single block of code is less susceptible to editing or automated merge errors than six blocks of nearly identical code (figure 3).

The two earlier blocks of code from `SSLVerifySignedServerKeyExchange()` now appear as follows:

```

if(isRsa) {
    /* ... */
    if ((err = HashHandshake(
        &SSLHashMD5, &clientRandom,
        &serverRandom,
        &signedParams, &hashOut))
        != 0)
        goto fail;
} else {...}
...
if ((err = HashHandshake(
    &SSLHashSHA1, &clientRandom,
    &serverRandom, &signedParams,
    &hashOut)) != 0)
    goto fail;

```

FIGURE 3

The handshake algorithm extracted into its own function

```

static OSStatus
HashHandshake(const HashReference* hashRef, SSLBuffer* clientRandom,
               SSLBuffer* serverRandom, SSLBuffer* exchangeParams,
               SSLBuffer* hashOut) {
    SSLBuffer hashCtx;
    OSStatus err = 0;
    hashCtx.data = 0;
    if ((err = ReadyHash(hashRef, &hashCtx)) != 0)
        goto fail;
    if ((err = hashRef->update(&hashCtx, clientRandom)) != 0)
        goto fail;
    if ((err = hashRef->update(&hashCtx, serverRandom)) != 0)
        goto fail;
    if ((err = hashRef->update(&hashCtx, exchangeParams)) != 0)
        goto fail;
    err = hashRef->final(&hashCtx, hashOut);
fail:
    SSLFreeBuffer(&hashCtx);
    return err;
}

```

This works because the HashReference is a “jump table” structure, and SSLHashMD5 and SSLHashSHA1 are instances of HashReference, which point to specific hash algorithm implementations. The HashReference interface makes it straightforward to write a small test exercising every path through the isolated HashHandshake() algorithm using a HashReference stub, and to verify that it would have caught this particular error:

```

+ build/libsecurity_ssl.build/.../
  x86_64/tls_digest_test
TestHandshakeFinalFailure failed:
  expected FINAL_FAILURE,
  received SUCCESS
1 test failed

```

The code for `tls_digest_test.c` is viewable at <http://goo.gl/tnvIUu> and contains all of my proof-of-concept changes; `build.sh` automates downloading the code, applying the patch, and building and running the test with a single

command. The test and the patch are very quick efforts but work as a stand-alone demonstration without requiring the full set of dependencies needed to build the entire library. The demonstration admittedly doesn't address further duplication or other issues present in the code.

The point of all this is, if an ex-programmer who has been out of the industry for two and a half years can successfully refactor and test this code within a couple of hours, never having seen it before, why didn't the engineer or team responsible for the code properly test it 17 months earlier?

WHY DIDN'T A TEST CATCH IT?

Several articles have attempted to explain why the Apple SSL vulnerability made it past whatever tests, tools, and processes Apple may have had in place, but these explanations are not sound, especially given the above demonstration to the contrary *in working code*. The ultimate responsibility for the failure to detect this vulnerability prior to release lies not with any individual programmer but with the culture in which the code was produced. Let's review a sample of the most prominent explanations and specify why they fall short.

Adam Langley's oft-quoted blog post¹³ discusses the exact technical ramifications of the bug but pulls back on asserting that automated testing would have caught it: "A test case could have caught this, but it's difficult because it's so deep into the handshake. One needs to write a completely separate TLS stack, with lots of options for sending invalid handshakes."

This "too hard to test" resignation complements the "I don't have time to test" excuse Google's Test Mercenaries, of which I was a member, often heard (though, by the time we disbanded, testing was well ingrained at Google, and the excuse was rarely heard anymore).¹¹ As already demonstrated, however, a unit test absolutely would have caught this, without an excess of difficulty. Effectively testing the algorithm does not require "a completely separate TLS stack"; a well-crafted test exercising well-crafted code would have caught the error—indeed, the *thought* of testing likely would have prevented it altogether.

Unfortunately, some adopted Langley's stance without considering that the infeasibility of testing everything at the system level is why the small, medium, and large test size schema (figure 4) exists (that's *unit*, *integration*, and *system* to most of the world outside Google).⁵ Automated tests of different sizes running under a continuous integration system (e.g., Google's TAP, Solano CI) are becoming standard practice throughout the industry. One would expect this to be a core feature of a major software-development operation such as Apple's, especially as it pertains to the security-critical components of its products.

Writing for *Slate*, David Auerbach breaks down the flaw for nonprogrammers and hypothesizes that the bug might have been caused by a merge error (based on this diff: <https://gist.github.com/alexyakoubian/9151610/revisions>; look for green line 631), but then concludes: "I think the code is reasonably good by today's standards. Apple wouldn't have released the code as open source if it weren't good, and even if they had, there would have been quite an outcry from the open-source community if they'd looked it over and found it to be garbage."³

Auerbach's conclusion assumes that everything Apple releases is high quality by definition, that it has every reasonable control in place to assure such high quality, and that all open-source code receives the focused scrutiny of large numbers of programmers (thanks to Stephen Vance for pointing this out specifically in his comments on my earlier presentation, which inspired this article)—at least, programmers motivated to report security flaws. The actual code, however, suggests a lack of automated testing discipline and the craftsmanship that accompanies it, as well as the absence of other quality

controls, not the fallibility of the existing discipline that Auerbach imagines Apple already applies.

Security guru Bruce Schneier notes, “The flaw is subtle, and hard to spot while scanning the code. It’s easy to imagine how this could have happened by error.... Was this done on purpose? I have no idea. But if I wanted to do something like this on purpose, this is exactly how I would do it.”¹⁵ Schneier’s focus is security, not code quality, so his perspective is understandable; but the evidence tips heavily in favor of programmer error and a lack of quality controls.

Delft University computer science professor Arie van Deursen notes many industry-standard tools and practices that could have caught the bug; but despite self-identifying as a strong unit-testing advocate, he demurs from asserting that the practice should have been applied: “In the current code, functions are long, and they cover many cases in different conditional branches. This makes it hard to invoke specific behavior.... Thus, given the current code structure, unit testing will be difficult.”¹⁶ As already demonstrated, however, this one particular, critical algorithm was easy to extract and test. Software structure can be changed to facilitate many purposes, including improved testability. Promoting such changes was the job of the Test Mercenaries at Google.

My former Test-Mercenary colleague C. Keith Ray noted both in his comments on van Deursen’s post and in his own blog: “Most developers who try to use TDD [test-driven development] in a badly designed, not-unit-tested project will find TDD is hard to do in this environment, and will give up. If they try to do ‘test-after’ (the opposite of TDD’s test-first practice), they will also find it hard to do in this environment and give up. And this creates a vicious cycle: untested bad code encourages more untested bad code.”¹⁴

I largely agree with Ray’s statement but had hoped he might seize the opportunity to mention the obvious duplicate code smell and how to eliminate it. Again, that was our stock-in-trade as Test Mercenaries. Absence of TDD in the past doesn’t preclude making code more testable now, and we have a responsibility to demonstrate how to do so.

Columbia University computer science professor Steven M. Bellovin provides another thorough explanation of the bug and its ramifications, but when he asks “why they didn’t catch the bug in the first place,” his focus remains on the infeasibility of exhaustive system-level testing: “No matter how much you test, you can’t possibly test for all possible combinations of inputs that can result to try to find a failure; it’s combinatorially impossible.”⁴

As demonstrated, this vulnerability wasn’t a result of insufficient system testing; it was because of insufficient unit testing. Keith Ray himself wrote a “Testing on the Toilet”⁸ article, “Too Many Tests,”¹¹ explaining how to break complex logic into small, testable functions to avoid a combinatorial explosion of inputs and still achieve coverage of critical corner cases (“equivalence class partitioning”). Given the complexity of the TLS algorithm, unit testing should be the first line of defense, not system testing. When six copies of the same algorithm exist, system testers are primed for failure.

Such evidence of a lack of developer testing discipline, especially for security-critical code, speaks to a failure of engineering and/or corporate culture to recognize the importance and impact of unit testing and code quality, and the real-world costs of easily preventable failures—and to incentivize well-tested code over untested code. Comments by an anonymous ex-Apple employee quoted by Charles Arthur in *The Guardian*² support this claim:

“Why didn’t Apple spot the bug sooner?”

“The former programmer there says, ‘Apple does not have a strong culture of testing or test-driven development. Apple relies overly on *dogfooding* [using its own products] for quality processes, which in security situations is not appropriate....’

“What lessons are there from this?”

“But the former staffer at Apple says that unless the company introduces better testing regimes—static code analysis, unit testing, regression testing—‘I’m not surprised by this... it will only be a matter of time until another bomb like this hits.’ The only—minimal—comfort: ‘I doubt it is malicious.’”

Reviewer Antoine Picard, commenting on the similarity between this security vulnerability and reported problems with Apple’s MacBook power cords, noted: “When all that matters is the design, everything else suffers.”¹²

HOW CAN WE FIX THE ROOT CAUSE?

Those with unit-testing experience understand its productivity benefits above and beyond risk prevention; but when the inexperienced remain stubbornly unconvinced, high-visibility bugs such as this can demonstrate the concrete value of unit testing—*in working code*.

Seize the teachable moments! Write articles, blog posts, flyers, give talks, start conversations; contribute working unit tests when possible; and hold developers, teams, and companies responsible for code quality.

Over time, through incremental effort, culture can change. The Apple flaw, and the Heartbleed bug discovered in OpenSSL in April 2014—after this article was originally drafted—could have been prevented by the same unit-testing approach that my Testing Grouplet (<http://mike-bland.com/tags/testing-grouplet.html>), Test Certified,⁶ Testing on the Toilet, and Test Mercenary partners in crime worked so hard to demonstrate to Google engineering over the course of several years. By the time we finished, thorough unit testing had become the expected cultural norm. (My commentary on Heartbleed, with working code, is available at <http://mike-bland.com/tags/heartbleed.html>.)

Culture change isn’t easy, but it’s possible. If like-minded developers band together across teams, across companies, even across the industry—such as is beginning to happen with the Automated Testing Boston Meetup (<http://www.meetup.com/Automated-Testing-Boston/>), its sister groups in New York, San Francisco, and Philadelphia, and the AutoTest Central community blog (<http://autotestcentral.com/>)—and engage in creative pursuits to raise awareness of such issues and their solutions, change will come over time.

The goal is that this and upcoming articles (including my “Goto Fail, Heartbleed, and Unit-testing Culture” article published by Martin Fowler, <http://martinfowler.com/articles/testing-culture.html>) will drive discussion around the Apple SSL and Heartbleed bugs, spreading awareness and improving the quality of discourse—not just around these specific bugs, but around the topics of unit testing and code quality in general. These bugs are a perfect storm of factors that make them ideal for such a discussion:

- The actual flaw is very obvious in the case of the Apple bug, and the Heartbleed flaw requires only a small amount of technical explanation.
- The unit-testing approaches that could have prevented them are straightforward.
- User awareness of the flaws and their severity is even broader than for other well-known software defects, generating popular as well as technical press.

- The existing explanations that either dismiss the ability of unit testing to find such bugs or otherwise excuse the flaw are demonstrably unsound.

If we don't seize these opportunities to make a strong case for the importance and impact of automated testing, code quality, and engineering culture, and hold companies and colleagues accountable for avoidable flaws, how many more preventable, massively widespread vulnerabilities and failures will occur? What fate awaits us if we don't take appropriate corrective measures in the wake of `goto fail` and Heartbleed? How long will the excuses last, and what will they ultimately buy us?

And what good is the oft-quoted bedrock principle of open-source software, Linus's Law—"Given enough eyeballs, all bugs are shallow"—if people refuse to address the real issues that lead to easily preventable, catastrophic defects?

I have worked to produce artifacts of sound reasoning based on years of experience and hard evidence—working code in the form of the Apple patch-and-test tarball and `heartbleed_test.c` (<http://goo.gl/w1bGyR>)—to back up my rather straightforward claim: a unit-testing culture most likely could have prevented the catastrophic `goto fail` and Heartbleed security vulnerabilities.

High-profile failures such as the Apple SSL/TLS vulnerability and the Heartbleed bug are prime opportunities to show the benefits of automated testing in concrete terms; to demonstrate technical approaches people can apply to existing code; and to illustrate the larger, often cultural, root causes that produce poor habits and bugs. Given the extent to which modern society has come to depend on software, the community of software practitioners *must* hold its members accountable, however informally, for failing to adhere to fundamental best practices designed to reduce the occurrence of preventable defects—and must step forward not to punish mistakes but to help address root causes leading to such defects. *If you see something, say something!*

ATTRIBUTION/FURTHER READING

This article is based on my presentation, "Finding More than One of the Same Worm in the Apple" (<http://goo.gl/FOURUR>), and the corresponding one-page Testing-on-the-Toilet-inspired treatment (<http://goo.gl/Lshkmj>). These were based on my blog entry, "Test Mercenary (Slight Return)" (<http://mike-bland.com/2014/03/04/test-mercenary-slight-return.html>), and my AutoTest Central article, "Finding the Worm Before the Apple Has Shipped" (<http://autotestcentral.com/finding-the-worm-before-the-apple-has-shipped>). Excerpts from my blog post, "The Official Apple SSL Bug Testing on the Toilet Episode" (<http://mike-bland.com/2014/04/15/goto-fail-tott.html>), were also used in the concluding section. All were published under a Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/deed.en_US).

The small, medium, and large test pyramid image shown in figure 4 is by Catherine Laplace, based on the author's sketch of an image from the Testing Grouplet/EngEDU Noogler Unit Testing lecture slides for new Google engineers.

PARTNERS IN CRIME

My deepest gratitude extends to my former Google colleagues, new associates from the Automated Testing Boston Meetup, and generous acquaintances whom I've met only online: David Plass, Isaac Truett, Stephen Vance, RT Carpenter, Gleb Bahmutov, Col Willis, Chris Lopez, and Antoine Picard. They provided tremendous input into the slides and one-page treatment, producing the structure and focus evident in those works and this article.

FIGURE 4

The Small/Medium/Large Test Strategy

I'd like to thank Sarah Foster of the Automated Testing Boston Meetup and the AutoTest Central blog for providing a forum to discuss this issue and the opportunity to connect with other like-minded developers.

Finally, I don't know how I'll ever repay Guido van Rossum of Python and Dropbox for advocating on my behalf that this article be published in *ACM Queue*, and Martin Fowler of ThoughtWorks for engaging me to write the "Goto Fail, Heartbleed, and Unit Testing Culture" article.

REFERENCES

1. Apple Inc. 2014. Xcode overview; https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/Xcode_Overview.pdf.
2. Arthur, C. 2014. Apple's SSL iPhone vulnerability: how did it happen, and what next? *The Guardian*, (February 25); <http://www.theguardian.com/technology/2014/feb/25/apples-ssl-iphone-vulnerability-how-did-it-happen-and-what-next>.
3. Auerbach, D. 2014. An extraordinary kind of stupid. *Slate* (February 25); http://www.slate.com/articles/technology/bitwise/2014/02/apple_security_bug_a_critical_flaw_was_extraordinarily_simple.html.
4. Bellovin, S. M. 2014. Goto Fail. *SMBlog* (February 23); <https://www.cs.columbia.edu/~smb/blog/2014-02/2014-02-23.html>.
5. Bland, M. 2014. AutoTest Central; <http://autotestcentral.com/small-medium-and-large-test-sizes>.

6. Bland, M. 2011. Test Certified; <http://mike-bland.com/2011/10/18/test-certified.html>.
7. Bland, M. 2012. Test Mercenaries; <http://mike-bland.com/2012/07/10/test-mercenaries.html>.
8. Bland, M. 2011. Testing on the Toilet; <http://mike-bland.com/2011/10/25/testing-on-the-toilet.html>.
9. Dijkstra, E. 1968. A case against the GO TO statement. *Communications of the ACM* 11 (3): 147–148; <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>.
10. Fuller, L. 2014. TestableSecurity: demonstrating that `SSLVerifySignedServerKeyExchange()` is trivially testable; <https://github.com/landonf/Testability-CVE-2014-1266>.
11. Google, Inc. 2008. Too many tests. *Google Testing Blog* (February 21); <http://googletesting.blogspot.com/2008/02/in-movie-amadeus-austrian-emperor.html>.
12. Greenfield, R. 2012. Why Apple's power cords keep breaking. *The Wire* (July 30); <http://www.thewire.com/technology/2012/07/why-apples-power-cords-keep-breaking/55202/>.
13. Langley, A. 2014. Apple's SSL/TLS bug. *Imperial Violet* (February 22); <https://www.imperialviolet.org/2014/02/22/applebug.html>.
14. Ray, C. K. 2014. TDD and signed `SSLVerifySignedServerKeyExchange`. *Exploring Agile Solutions: Software Development with Agile Practices* (February 23); <http://agilesolutionspace.blogspot.com/2014/02/tdd-and-signed-sslverifysignedserverkey.html>.
15. Schneier, B. 2014. Was the iOS SSL flaw deliberate? *Schneier on Security: A Blog Covering Security and Security Technology* (February 27); https://www.schneier.com/blog/archives/2014/02/was_the_ios_ssl.html.
16. van Deursen, A. 2014. Learning from Apple's #gotofail security bug. *Arie van Deursen: Software Engineering in Theory and Practice* (February 22); <http://avandeursen.com/2014/02/22/gotofail-security/>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

MIKE BLAND was a software engineer at Google from 2005 to 2011. Prior to working on Web-search infrastructure, he led the Testing and Fixit Grouplets; was a member of the Test Mercenaries, Testing Tech, and Build Tools teams; and was instrumental in bringing about the engineering culture changes that made thorough developer testing the accepted cultural norm. He does not represent Google in any capacity and is a student at Berklee College of Music. <http://mike-bland.com/>.

© 2014 ACM 1542-7730/14/0500 \$10.00