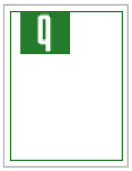
Security
Vol. 12 No. 5 – May 2014



Finding More Than One Worm in the Apple
If you see something, say something.

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.

by Mike Bland

Who Must You Trust?
You must have some trust if you want to get anything done.

In his novel The Diamond Age, author Neal Stephenson describes a constructed society (called a phyle) based on extreme trust in one's fellow members. Part of the membership requirements is that, from time to time, each member is called upon to undertake certain tasks to reinforce that trust. For example, a phyle member might be told to go to a particular location at the top of a cliff at a specific time, where he will find bungee cords with ankle harnesses attached. The other ends of the cords trail off into the bushes. At the appointed time he is to fasten the harnesses to his ankles and jump off the cliff. He has to trust that the unseen fellow phyle member who was assigned the job of securing the other end of the bungee to a stout tree actually did his job; otherwise, he will plummet to his death. A third member secretly watches to make sure the first two don't communicate in any way, relying only on trust to keep tragedy at bay. Whom you trust, what you trust them with, and how much you trust them are at the center of the Internet today, as well as every other aspect of your technological life.

by Thomas Wadlow

Automated QA Testing at EA: Driven by Events
A discussion with Michael Donat, Jafar Husain, and Terry Coatta

To millions of game geeks, the position of QA (quality assurance) tester at Electronic Arts must seem like a dream job. But from the company's perspective, the overhead associated with QA can look downright frightening, particularly in an era of massively multiplayer games.

by Terry Coatta, Michael Donat, Jafar Husain

acmqueue

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;
        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/tnvIUm 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 post[13] 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

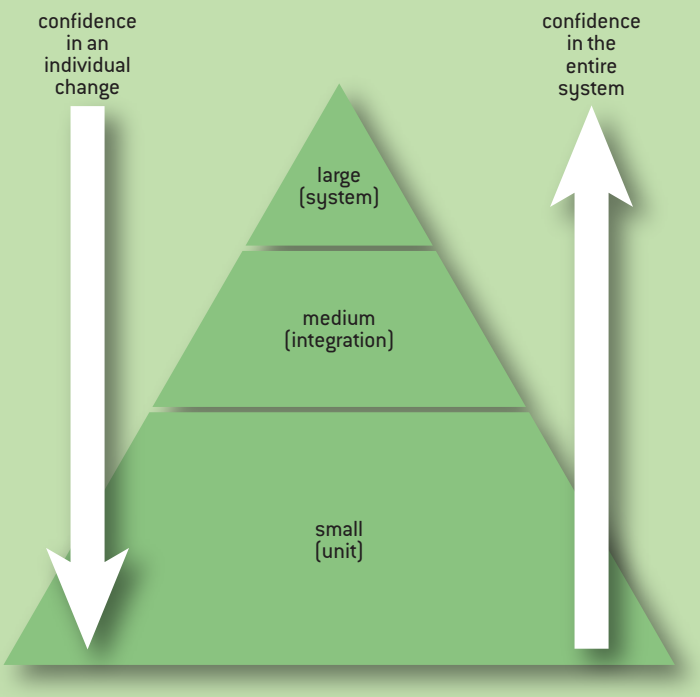
This article is based on my presentation, "Finding More than One of the Same Worm in the Apple" (http://goo.gl/F0URUR), and the corresponding one-page Testing-on-the-Toilet-inspired treatment (http://goo.gl/Lshkmj). These were based on my blog entry, "Test Mercenary (Slight Return)" (http://mike-bland.com/2014/03/04/test-mercenary-slight-return.html), 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/.

acmqueue

Who Must You Trust?

You must have some trust if you want to get anything done

Thomas Wadlow

In his novel *The Diamond Age*,[7] author Neal Stephenson describes a constructed society (called a *phyle*) based on extreme trust in one's fellow members. Part of the membership requirements is that, from time to time, each member is called upon to undertake certain tasks to reinforce that trust. For example, a phyle member might be told to go to a particular location at the top of a cliff at a specific time, where he will find bungee cords with ankle harnesses attached. The other ends of the cords trail off into the bushes. At the appointed time he is to fasten the harnesses to his ankles and jump off the cliff. He has to trust that the unseen fellow phyle member who was assigned the job of securing the other end of the bungee to a stout tree actually did his job; otherwise, he will plummet to his death. A third member secretly watches to make sure the first two don't communicate in any way, relying only on trust to keep tragedy at bay.

Whom you trust, what you trust them with, and how much you trust them are at the center of the Internet today, as well as every other aspect of your technological life.

TRUST IN HARDWARE

During the race to the moon in the 1960s, the Apollo program was faced with the unprecedented problem of guiding two manned spacecraft to a rendezvous in lunar orbit.[4] Because of the speed-of-light delay in radio transmissions to and from the moon, guidance from ground-based computers would have an unacceptable delay from anything close to real-time, endangering the mission and the lives of the astronauts. A better answer was to have on-board computation with minimal lag time to help the pilots determine how to rendezvous the two spacecraft.

At that time, computers filled rooms and weighed tons. In order to build computer systems that were small and lightweight enough to fly with the Apollo Command and Lunar Modules, NASA engineers wanted to turn to the newly developed integrated circuit chips. The problem: these early chips were not particularly reliable even for ground uses, let alone for mission-critical spacecraft flight hardware.

In the pre-microprocessor 1960s, computer systems were assembled from individual components. At first these were individual transistors, but with chips the components could become gates and flip-flops, registers and arithmetic logic units. But all of these chips were unreliable—especially, but not exclusively, under spaceflight conditions, where extremes of pressure, temperature, vibration, radiation, and high and low gravity added to the potential problems.

Chip reliability could be developed, but perfecting the reliability of even a single chip required enormous R&D, as well as high-volume purchases. Developing the whole suite of reliable chips needed to build a flight-capable computer system appeared to be beyond even NASA's resources.

NASA's solution was simple, brilliant, and exploited the subtleties of computer logic. Rather than trying to perfect dozens of chip types, NASA selected a single chip design that would contain two

3-input NOR gates per chip. Boolean logic teaches us that all logic functions can be built up from NOR gates. By playing to the strengths of Boolean logic, the flight computer was designed using just this single type of chip. A great deal of specialized ground equipment was designed using the same chip as well, so that lesser-than-flight-quality chips could still be used productively. This practice ensured large-volume purchases of the one type of chip, which made it worthwhile for both NASA and the chipmakers to devote huge resources to perfecting the reliability of that chip. With one trustworthy chip as the basis for the flight computers, no Apollo mission suffered even a single computer component failure.

The Space Shuttle went one step further. As one of the first fly-by-wire avionics systems, when such systems were not well trusted, it used five computers that voted on flight-control actions. Four computers ran identical software; the fifth ran software written by a different vendor, but with identical specifications, to guard against a bug in the primary software package. If one of the primary computers disagreed with the other three and the fifth, it could be isolated and ignored. The system was layered and redundant in both hardware and software.

Hardware can fail for any number of reasons, including inadequate testing, design faults, component faults, power-supply issues, ground loops, interface voltage mismatches, externally induced power surges, and human error in maintenance. It can also be untrustworthy because of HTs (hardware Trojans),[9] in which a device is designed or modified to behave in a manner benefitting the Trojan designer, not the purchaser or operator of the hardware. Detecting a hardware Trojan can be fiendishly difficult, and reverse engineering the intentions of an HT if found is even more so.

TRUSTING SOFTWARE

Technologists tend to prefer technological solutions. If there's a problem, then perhaps there's a box, or a piece of software, that can be bought or built to solve it. This can often work well for problems of physics, or logic, or regular organization. File servers hold files; backup servers back them up. Atomic clocks measure time; routers move packets between networks.

That same regularity is not true of security. It's critically important to remember that security is about people, not technologies. Many people act honorably, but some make mistakes, and some lie, cheat, and steal—sometimes for their own profit or advantage; other times to demonstrate a system's vulnerability. People cause their technologies to act for them in a like manner.

Ken Thompson, co-inventor of the C programming language and the Unix operating system, demonstrated this issue in a memorable way[10] in 1984. He created a C program fragment that would introduce Trojan Horse code into a program compiled by the C compiler. For example, when compiling the program that accepts passwords for login, you could add code that would cause the program to accept legitimate passwords or a special backdoor password known to the creator of the Trojan. This is a common strategy even today and is often detectable through source-code analysis.

Thompson went one step further. Since the C compiler is written in the C programming language, he used a similar technique to apply a Trojan to the C compiler source itself. When the C compiler is compiled, the resulting binary program could be used to compile other programs just as before; but when the program that accepts passwords for login is compiled with the new compiler from clean, uncompromised source code, the backdoor-password Trojan code is inserted into the binary, even though the original source code used was completely clean. Source-code analysis would not reveal the Trojan because it was lower in the tool chain than the login program.

He then went on to point out that this technique could be applied even lower still, at the assembler, loader, or hardware microcode level.

Thompson's moral was: "You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me)."

TRUSTING THE NETWORK

In the late 1700s, a Frenchman named Claude Chappe[3] invented a visual telegraph that used semaphore towers spaced 10 to 15 kilometers apart. With a telescope, each station could see neighboring towers and relay their semaphore positions to the destination.

In a world where long-distance communication was via horse-carried letters, Chappe's telegraph was a wonder. Signals from Paris were received in Lille, where the first stations were built (about 218 kilometers or 136 miles apart), after only a few minutes. The network soon spread to Brest, Toulon, Strasbourg, and later even across the English Channel.

Semaphore positions were defined in a codebook allowing a variety of messages to be sent. The well-crafted codebook provided easily recognizable semaphore patterns for common symbols, as well as error-correction codes that allowed a proper message to emerge despite errors in transmission or reception.

The Chappe network was in continuous use for more than 60 years. An electrical telegraph network supplanted it once the technology of long-distance insulated wires made that invention practical.

For our purposes, though, the most interesting aspect of the Chappe network came about in 1836 with the discovery that stock market information was being relayed across the network, buried in other messages, by means of what we now might call steganography. The semaphore operators were paid to introduce certain errors in the messages of various customers. Because of the error-correction codes the messages arrived intact, but if someone was privy to the raw symbols transmitted, the introduced errors contained a message that provided an advantage in buying or selling stocks.

A network can seem trustworthy but still be used for purposes other than its intended one.

Even the time of day can be exploited. In 2013 a network attack known as NTP Amplification used Network Time Protocol servers across the Internet in a distributed denial-of-service attack. By spoofing the IP address of a requester, an ever-larger stream of packets could be aimed at a target, swamping the target's ability to respond to TCP/IP requests.

EXPLORING TRUST

Here's an experiment to try. Take walks in various mixed-use neighborhoods, each with a variety of residences and businesses, such as restaurants, cafes, hardware stores, and hairdressers. Walk in the daytime, before and after lunch. Walk in the nighttime, at the height of the evening activities. Walk late at night, after most things have shut down. With each outing, put yourself in a security mindset. Which is to say: look with the eyes of a thief and notice what you see.

During the day, for example, at busy sidewalk cafes, do people reserve outdoor tables by placing their possessions on the table and then going inside to order? Do they use their grocery bags for this? Their car and house keys? Their wallets?

Late at night, are those same tables and chairs stacked outside or inside? Are they chained together? Are the chains lightweight or substantial? Do the homes in the neighborhood have porch furniture or lawn tools visible from the street? Are they locked up?

Do you see bars on the windows of the residences? On all the windows, or just the back-alley windows? Are there possessions stored on the lawns, in the carports, or on the porches outside at night? Are the family cars parked outside? Are the car windows open? Do they have steering-wheel locks?

Do you see loading docks, such as those in the back of a grocery store, with open doors but nobody visible?

In a district with nightclubs, do you see people who are visibly drunk on the street late in the evening?

Do kids walk to school unchaperoned?

Do postal workers or delivery services leave packages unattended by the front doors of houses? Are bundles of newspapers and magazines left in front of newsstands before they open?

You can learn a lot about neighborhoods this way. It's an especially interesting exercise if you are shopping for real estate. These observations, and many more, are flags for the implicit levels of trust that people have in their neighbors and neighborhoods. The people themselves may not even think of these things. They may leave things on their porches, perhaps accidentally, and nothing bad happens, so they don't worry if it happens again. After a while, it becomes something they don't even notice that they do.

TRUST ON THE INTERNET

Walking around a physical neighborhood, you can gather a lot of information if you are open to it. On the Internet, it can be a very different story. Imagine an office with a single PC, used in the morning by Albert and in the afternoon by Betty. Each has a different account and logs out at the end of the shift. Albert and Betty work from the same physical place, and from the same IP address, but they may have very different experiences on the Internet, depending on what they do.

Extend that to a data center hosting many companies. Each company's servers may be separated by only a few feet, but how they experience the Internet, and how the Internet experiences them, can vary widely. The physical distance between the servers is irrelevant. What matters is the hardware they are composed of, the operating systems that run on them, the application server software, and the configuration data for each of those things, plus all of the utility and ancillary software needed to support and maintain them. The quality and quantity of users, as well as administrators, also matter a great deal.

HEARTBLEED

In spring 2014, a bug in the open-source package OpenSSL became widely known. The bug, now known as Heartbleed (http://heartbleed.com), had been present for some time, and may have been known by some, but the full disclosure of the problem in the OpenSSL package came to the public's attention only recently. OpenSSL had been reviewed by many experts and had been a well-used and trusted part of the Internet ecosystem until that point. As of this writing, there is no evidence suggesting any cause other than a programming error on the part of an OpenSSL contributor.

On the morning before the Heartbleed bug was made public, few people were familiar with OpenSSL and they hardly gave the functions it provided a second thought. Those who knew of it often had a strong level of trust in it. By the end of the day, that had all changed. Systems administrators and companies of all sizes were scrambling to contain the problem. Within just a

few days, this obscure piece of specialized software was at the top of the news cycle, and strangers—perhaps sitting in outdoor cafes at tables they had reserved with their house and car keys—were discussing it in the same tones with which they might have discussed other catastrophes.

SYSTEMS ADMINISTRATORS

At the heart of everything that works on the Internet are systems administrators. Sometimes they are skilled experts, sometimes low paid and poorly trained, sometimes volunteers of known or unknown provenance. Often they work long, unappreciated hours fixing problems behind the scenes or ones that are all too visible. They have access to systems that goes beyond that of regular users.

One such systems administrator worked for the NSA (National Security Agency). His name is Edward Snowden. You probably know more about him now than you ever expected to know about any sysadmin, even if you are one yourself.

Another less familiar name is Terry Childs,[5,11] a network administrator for the city of San Francisco, who was arrested in 2008 for refusing to divulge the administrative passwords for the city's FiberWAN network. This network formed the core of many city services. According to reports, Childs, a highly qualified and certified network engineer who designed and implemented much of the city's network himself, was very possessive of it—perhaps too possessive, as he became the sole administrator of the network, claiming not to trust his colleagues' abilities. He allowed himself to be on-call 24/7, year-round, rather than delegate access to those he considered less qualified.

After an argument with a new boss who wanted to audit the network against Childs's wishes, the city's CIO demanded that Childs provide the administrative credentials to the FiberWAN. Childs refused, which led to his arrest. Even after his arrest, Childs would not provide administrative access to the network. Finally he relented and gave the mayor of San Francisco the access credentials, ending the standoff.

His supervisors claimed he was crazy and wanted to damage the network. Childs claimed he did not want to provide sensitive access credentials to unqualified individuals who might damage "his" network.

In 2010, Childs was found guilty of felony network tampering and sentenced to four years in prison and $1.5 million in restitution for the costs the city incurred in regaining control of the network. An appeals court upheld the verdict.

Was Childs a fanatic, holding on too tight for his own good, or a highly responsible network admin who would not allow his network to be mismanaged by people he considered to be incompetent? Consider these questions:

- Could something like this happen at your enterprise? How would you know that this problem was developing, *before* it became a serious problem?
- What safeguards do you have in place to prevent a single-point concentration of power such as this?
- What would you do if your organization found itself in this situation?

WHO MUST YOU TRUST?

Some people dream of going back to nature and living apart from the rest of humanity. They want to build their own cabins, grow or raise their own food, and live entirely off infrastructure they have built with their own two hands and a trusty ax. But who made that ax? Even if you can make a

hand-chipped flint ax from local materials, it is far from "trusty," and the amount of wood you can cut with a flint ax pales in comparison to what you can cut with a modern steel ax. So if you go into the woods with a modern ax, can you truly say you are independent of the world?

If you work on the Internet, or provide some service to the Internet, you have a similar problem. You cannot write all of the code if you intend to provide a modern and useful network service. Network stacks, disk drivers, Web servers, schedulers, interrupt handlers, operating systems, compilers, software-development environments, and all the other layers needed to run even a simple Web server have evolved over many years. To reinvent it all from the specifications, without using other people's code anywhere in the process, is not a task for the faint-hearted. More importantly, you couldn't trust it completely even if you did write it all. You would be forever testing and fixing bugs before you were able to serve a single packet, let alone a simple Web page.

Neither can you build all of the hardware you run that service on. The layers of tools needed to build even a simple transistor are daunting, let alone the layers on top of that needed to build a microprocessor. Nor can you can build your own Internet to host it. You have to trust some of the infrastructure necessary to provide that service. But which pieces?

HOW MUCH DO YOU NEED TO TRUST?

To determine how far your trust needs to extend, start with an evaluation of your service and the consequences of compromise. Any interesting service will provide some value to its users. Many services provide some value to their providers. What is valuable about your service, and how could that value be compromised?

Once you have a handle on those questions, you can begin to think about the minimum of components and services needed to provide such a service and which components you have to trust.

Writing your own software can be part of this exercise, but consider that the bulk of security derived from that is what is known as "security through obscurity." Attacks will fail because attackers don't understand the code you have built—or so some think. If you choose the path of obscurity as a strategy, you're betting that no one will show interest in attacking your service, that your programmers are better than others at writing obscure code in a novel way, and that even if the code is obscure, it will still be secure enough that someone determined to break through it will be thwarted. History has shown that these are not good bets to make.

WHO IS EVERYBODY ELSE TRUSTING?

A better approach might be to survey the field to see what others in similar positions are doing. After all, if most of your competitors trust a particular software package to be secure, then you are all in the same situation if it fails. There are variables, of course, because any software, even the best, can be untrustworthy if it is badly installed or configured. And your competitors might be mistaken.

A variation on this approach is to find out which software all of your competitors *wish* they could use. Moving to what they use now could leave you one generation behind by the time you get it operational. On the other hand, moving to one generation ahead could leave you open to yet-undetected flaws. The skill is in choosing wisely.

HOW ARE THOSE SERVICES EVALUATED FOR SECURITY?

Whatever components or services you choose, consider how they have been tested for

trustworthiness. Consider these principles attributed to Auguste Kerckhoffs, a Dutch linguist and cryptographer, in the 19th century:

- The system should be, if not theoretically unbreakable, then unbreakable in practice.
- The design of a system should not require secrecy, and compromise of the system design should not inconvenience the correspondents.

Kerckhoffs was speaking of cipher design in cryptosystems, but his two principles listed here can be applied to many security issues.

When considering components for your enterprise, do they live up to Kerckhoffs's principles? If they seem to, who says that they do? This is one of the strongest cases for open-source software. When done properly, the quality and security of open-source code can rival that of proprietary code.[2]

For services that you wish to subscribe to, consider how often and how thoroughly they are audited, and who conducts the audits. Do the service providers publish the results? Do they allow prospective customers to see the results? Do the results show their flaws and describe how they were fixed or remediated, or do they just give an overall thumbs-up?

THINKING ABOUT THE BAD CASES FIRST

The legendary Fred Brooks, he of *The Mythical Man Month,*[1] famously said: "All programmers are optimists." Brooks meant this in terms of the tendency of programmers to think that they can complete a project faster than it will actually take them to do it. But as ACM's own Kode Vicious is wont to point out, there is a security implication here as well. Developers often code the cases that they want to work first and, if there's enough time, fill in the error-handling code later, if at all.

When you are worried about security issues, however, reversing the order of those operations makes a lot of sense. If, for example, your application requires a cryptographic certificate to operate, one of the first issues a security programmer should think about is how that certificate can be revoked and replaced. Selecting certificate vendors from that perspective may be a very different proposition from the usual criteria (which almost always emphasize cost). Building agile infrastructure from the start, in which the replacement of a crypto cert is straightforward, easy to do, and of minimal consequence to the end user, points the way toward a process for minimizing trust in any one vendor.

Developing an infrastructure that makes it easy to swap out certificates leads to the next interesting question: How will you know when to swap out that bad certificate? Perhaps the question can be turned around: How expensive is it to swap out a certificate: in money, effort, and customer displeasure? If it can be done cheaply, quickly, easily, and with no customer notice, then perhaps it should be done frequently, just in case. If done properly, then a frequent certificate change would help limit the scope of any damage, even if a problem is not noticed at first.

But here there be dragons! Some might read the previous paragraph and think that having certificates that expire weekly, for example, eliminates the need to monitor the infrastructure for problems, or the need to revoke a bad certificate. Far from it! All of those steps are necessary as well. Security is a belt-and-suspenders world.

An infrastructure that is well monitored for known threats is another part of the trust equation. If you are confident that your infrastructure and personnel will make you aware of certain types of problems (or potential problems), then you can develop and practice procedures for handling those problems.

That covers the "known unknowns," as former Secretary of Defense Donald Rumsfeld[6] said, but what about the "unknown unknowns"? For several years Heartbleed was one of these. The fault in OpenSSL was present and exploitable for those who knew of it and knew how to do so. As of this writing, we do not know for certain if anybody did exploit it, but had someone done so, the nature of the flaw is such that an exploit would have left little or no trace, so it is very difficult to know for sure.

There are two major kinds of "unknown unknowns" to be aware of when providing a network service. The first are those unknowns that you don't know about, but somebody else might know about and have disclosed or discussed publicly. Let's call them "discoverable unknowns." You don't know about them now, but you can learn about them, either from experience or from the experiences of others.

Discoverable unknowns are discoverable if and only if you make the effort to discover them. The pragmatic way to do this is to create an "intelligence service" of your own. The Internet is full of security resources if you care to use them. It is also full of misdirection, exaggeration, and egotism about security issues. The trick is learning which resources are gold and which are fool's gold. That comes with practice and, sadly, often at the cost of mistakes both big and small.

A prudent, proactive organization has staff and budget devoted to acquiring and cultivating security resources. These include someone to evaluate likely Web sites, as well as read them regularly; subscriptions to information services; membership in security organizations; travel to conferences; and general cultivation of good contacts. It also includes doing favors for other organizations in similar situations and, if possible, becoming a good citizen and participant in the open-source world. If you help your friends, they will often help you when you need it.

The second type of unknowns can be called "unexpected unknowns." You don't know what they are, you don't even know for sure that they exist, so you are not on the lookout for them specifically. But you can be on the lookout for them in general, by watching the behavior of your network. If you have a way of learning the baseline behavior of your network, system, or application, then you can compare that baseline to what the system is doing now. This could include monitoring servers for unexpected processes, unexpected checksums of key software, files being created in unusual places, unexpected load changes, unexpected network or disk activity, failed attempts to execute privileged programs, or successful attempts that are out of the ordinary. For a network, you might look for unusual protocols, unexpected source or destination IP addresses, or unusually high- or low-traffic profiles. The better you can characterize what your system is supposed to be doing, the more easily you can detect when it is doing something else.

Detecting an anomaly is one thing, but following up on what you've detected is at least as important. In the early days of the Internet, Cliff Stoll,[8] then a graduate student at Lawrence Berkeley Laboratories in California, noticed a 75-cent accounting error on some computer systems he was managing. Many would have ignored it, but it bothered him enough to track it down. That investigation led, step by step, to the discovery of an attacker named Markus Hess, who was arrested, tried, and convicted of espionage and selling information to the Soviet KGB.

Unexpected unknowns might be found, if they can be found at all, by reactive means. Anomalies must be noticed, tracked down, and explained. Logs must be read and understood. But defenses against known attacks can also prevent surprises from unknown ones. Minimizing the "attack surface" of a network also minimizes the opportunities an attacker has for compromise.

Compartmentalization of networks and close characterization of regular traffic patterns can help detect something out of the ordinary.

WHAT CAN YOU DO?

How can issues of trust be managed in a commercial, academic, or industrial computing environment?

The single most important thing that a practitioner can do is to give up the idea that this task will ever be completed. There is no device to buy, no software to install, and no protocol to implement that will be a universal answer for all of your trust and security requirements. There will *never* come a time when you will be done with it and can move on to something else.

Security is a process. It is a martial art that you can learn to apply by study, thought, and constant practice. If you don't drill and practice regularly, you will get rusty at it, and it will not serve you when you need it. Even if you do become expert at it, an attacker may sometimes overpower you. The better you get at the process, however, the smaller the number of opponents that can do you harm, the less damage they can do, and the quicker you can recover.

Here are some basic areas where you can apply your efforts.

KNOW WHOM YOU TRUST AND WHAT YOU TRUST THEM TO DO

Though it is an overused phrase, "Web of Trust" is descriptive of what you are building. Like any sophisticated construction, you should have a plan, diagram, or some other form of enumeration for which trust mechanisms are needed to support your enterprise. The following entities might be on such a plan: data-center provider (power, A/C, LAN); telecommunications link vendors; hardware vendors; paid software vendors; open-source software providers; cryptographic certificate suppliers; time-source suppliers; systems administrators; database administrators; applications administrators; applications programmers; applications designers; and security engineers.

Of course, mileage may vary, and there may be many more entities as well. Whatever is on the list you generate, perform the following exercise for each entry:

- Determine whom this entity trusts to do the job and who trusts this entity.
- Estimate the consequences if this entity were to fail to do the job properly.
- Estimate the consequences if this entity were a bad actor trying to compromise the enterprise in some way (extract information without authorization, deny service, provide bad information to your customers or yourself, etc.).
- Rate each consequence for severity.

KNOW WHAT YOU WOULD DO IF ANY OF THOSE ENTITIES LOST YOUR TRUST

Now that you have a collection of possible ways that your enterprise can be affected, sorted by severity, you can figure out what you would do for each item. This can be as simple or complicated as you are comfortable with, but remember that you are creating a key part of your operations handbook, so if your plans cannot be turned into actions when these circumstances occur, they will not be worth much.

Here are some examples of the kinds of consequences and actions that might be needed:

- A key open-source package is discovered to have a serious bug and must be replaced with a newer, bug-fixed version; replaced with a different package with the same API; replaced with a different

package with a different API; or mitigated until a fix can be developed. Your plan should be a good guide to handling any of these situations.

- A key systems administrator has been providing network access to a potentially unfriendly third party. You must: determine the extent of information lost (or was your information modified?); determine if any systems were compromised with backdoor access; determine which other systems under the sysadmin might be affected; and figure out the best way of handling the personnel issues (e.g., firing, transfer, or legal action).
- A key data center is rendered unusable by a disaster or attack. You must: shift to a standby reserve location; or improvise a backup data center.

PRACTICE, PRACTICE, PRACTICE

Having a plan is all very nice, but if it's in a dusty file cabinet, or worse yet, on a storage volume in a machine that is made unavailable by the circumstances you are planning for, then it doesn't help anybody. Even if the plan is readily available, carrying it out for the first time during a crisis is a good way to ensure that it won't work.

The best way to make sure that your plan is actionable is to practice. That means every plan needs to have a method of simulating the cause and evaluating the result. Sometimes that can be as easy as turning off a redundant server and verifying that service continues. Other problems are more complex to simulate. Even a tabletop exercise, in which people just talk about what is needed, is better than never practicing your contingency plan.

Practice can also take the form of regular operations. For example, Heartbleed required many service providers to revoke and reissue certificates. If that is a critical recovery operation for your enterprise, then find a way to work that procedure into your regular course of business, perhaps by revoking and reissuing a certificate once a month.

Other operations can also benefit from practice, such as restoring a file from backups; rebuilding an important server; transferring operations to a backup data center; or verifying the availability of backup power and your ability to switch over to it.

SET MOUSETRAPS

The most important step in defending against attackers (or Murphy's Law) is learning that you have a problem. If you understand your trust relationships—who is trusted with what and who is not trusted—then watching for violations of those relationships will be very instructive. Every violation will probably fall into one of these categories:

- An undocumented but legitimate trust relationship. This might be sysadmins doing their assigned work, for example, but that work was improperly overlooked when building the trust map.
- A potentially reasonable but unconsidered potential trust relationship that must be evaluated and either added to the trust map or explicitly prohibited—for example, a sysadmin doing unassigned but necessary work to keep a system operational.
- An unreasonable or illegitimate use.

The only way to know which case it is will be to investigate each one and modify your trust map accordingly. As with all things of this nature, mousetraps must be periodically tested to see if they still work.

VET YOUR KEY PEOPLE

Often, trusting a systems administrator takes the form of management saying to sysadmins "Here are the keys to everything," followed by more-or-less blind trust that those keys will not be abused. Or, to quote science fiction author Robert Heinlein: "It's amazing how much mature wisdom resembles being too tired." That sort of blind trust is asking for trouble.

On the other hand, tracking sysadmins closely and forcing them to ask permission for every privileged operation they wish to perform can hobble an organization. Chances are good that both the sysadmins and the granters of permission will grow tired of this, and the organization will move back toward blind trust.

A good way to navigate between these two rocky shoals is to hire good people and treat them well. Almost as important is communicating with them to reinforce the security and trust goals of your organization. If they know what must and must not be done and, at least in general principle, why those constraints are good, then the chances are greater that they will act appropriately in a crunch.

LOG WHAT THEY DO. HAVE SOMEBODY ELSE REVIEW THOSE LOGS REGULARLY

Good people can make mistakes and sometimes even go astray. A regular non-privileged (in the security sense) employee should have a reasonable expectation of workplace privacy, but a systems administrator should know that he or she is being watched when performing sensitive tasks or accessing sensitive resources. In addition, sysadmins should perform extremely sensitive tasks with at least one other person of equal or higher clearance present. That way, someone else can attest that the action taken was necessary and reasonable.

Wherever possible, log what the sysadmins do with their privileges and have a third party review those logs regularly for anomalies. The third party should be distant enough from the systems administrators or other employees given trusted access that no personal or professional relationships will obscure the interpretation of the logs.

Investigate what you suspect and act on what you find. Let your trusted people know in advance that that is what you will do. Let them know that their positions of responsibility make them the first suspects on the list if trust is violated.

MINIMIZE YOUR WINDOWS OF VULNERABILITY

Once you know the ways in which you can be vulnerable, develop plans to minimize and mitigate those vulnerabilities. If you can close the hole, then close it. If you can't close it, then limit what can be done through the hole. If you can't limit what can be done, then limit who can do it and when it can be exploited. If you can't limit anything, then at least measure whether an exploit is taking place. You may not have a perfect solution, but the more limits you put on a potential problem, the less likely it is that it will become a real problem.

LAYER YOUR SECURITY

When it comes to trust, you should not depend on any one entity for security. This is known as "defense in depth." If you can have multiple layers of encryption, for example, each implemented differently (one depending on OpenSSL, for example, and the other using a different package), then a single vulnerability will not leave you completely exposed.

This is good reason to look at every component of your enterprise and ask: What if these components were to be compromised?

PRACTICE BEING AGILE

If a component were compromised, how would you replace it, and with what? How long would it take to switch over? Theories don't count here. You need to be prepared to switch packages or vendors or hardware in order to be adequately safe. How long will it take your purchasing department to cut paperwork for a new license, for example? How long to get that purchase order signed off? How long for the vendor to deliver?

This is not work you can do once and think you are ready. You need to revisit all components regularly and perform this kind of analysis for each of them as circumstances change.

LOOK AT YOUR NETWORK AS AN ATTACKER WOULD

Know the "as-built" configuration of your network, not just the "as-specified." Remember that the as-built configuration can change every day. This means you have to have people to measure the network, and tools to examine it. What network services does each component provide? Are those services needed? Are they available only to the places they are needed? Are all of the components fully patched? Are they instrumented to detect and report attack attempts? Does someone read the logs? What is the longest period of time between when an attack happens and when somebody notices it? Are there any events (such as holidays) when the length of time an attack goes unnoticed might increase?

The Internet abounds with free or inexpensive software for security analysis. These are tools often used by attackers and defenders. There is something to be learned by looking at your network through the same tools that your attackers use.

TRACK SECURITY ISSUES AND CONFIRM THAT THEY GET FIXED

If you find a problem, how is it tracked? Who is responsible for getting it into the tracking system, getting it to someone who can fix it, and getting it fixed? How do you measure that the problem is present? Do you measure again after the fix is applied to ensure that it worked?

DEVELOP YOUR OWN SECURITY INTELLIGENCE RESOURCES

Does your organization have personnel who track the technology used for potential security issues? How often do they check? Are they listened to when they report a problem?

Any equipment, software, vendors, or people you depend on should be researched on a regular basis. Quality security-focused Web sites exist, but they are often surrounded and outnumbered by those with products to sell or misinformation to distribute. Having staff gain the expertise to distinguish the good from the bad is extremely valuable.

PLAN FOR BIG-TICKET PROBLEMS

If you run a networked enterprise, whether you provide a public, private, or internal suite of services, you will find that trusted services *will* fail you, sooner or later. Repeatedly. How you respond to those failures of trust will become a big part of your company's reputation. If you select your vendors, partners, and components wisely, seriously plan for responses to trouble, and act on your plans when the time comes, then you will fare much better in the long run than those whose crisis planning is filed under "Luck."

CONCLUSION

The problem of trust is not new. If anything, the only *new* part is the mistaken impression that things can be trusted, because so many new things seem to be trustworthy. It is a sometimes-comforting illusion, but an illusion nonetheless. To build anything of value, you will have to place your trust in some people, products, and services. Placing that trust wisely is a skill that is best learned over time. Mistakes will abound along the way. Planning for your mistakes and the mistakes of others is essential to trusting.

It is generally better, faster, and safer to take something that meets good standards of trustworthiness and add value to it—by auditing it, layering on top of it, or adding to the open source—than it is to roll your own. Be prepared to keep a wary eye on the components you select, the system you include them in, and the people who build and maintain that system. Always plan for trouble, because trouble will surely come your way.

You must have some trust if you want to get anything done, but you cannot allow yourself to be complacent. Thomas Jefferson said, "Eternal vigilance is the price of liberty." It is the price of security as well.

ACKNOWLEDGMENTS

Thanks to Jim Maurer at ACM for requesting this article and George Neville-Neil for reminding me of the Frederick Brooks quote. Special thanks to my wife, Yuki, and my kids for putting up with me as I grumbled to myself while writing.

REFERENCES

1. Brooks Jr., F. P. 1975. *The Mythical Man-Month*. Addison-Wesley.
2. Coverity. 2013. Coverity Scan report finds open source software quality outpaces proprietary code for the first time; http://www.coverity.com/press-releases/coverity-scan-report-finds-open-source-software-quality-outpaces-proprietary-code-for-the-first-time/.
3. Dilhac, J.-M. The telegraph of Claude Chappe—an optical telecommunication network for the XVIIIth century: 7; http://www.ieeeghn.org/wiki/images/1/17/Dilhac.pdf.
4. Hall, E. C. 1996. Journey to the moon: the history of the Apollo guidance computer. Reston, VA: AIAA.
5. McMillan, R. 2008. IT admin locks up San Francisco's network. *PCWorld*; http://www.pcworld.com/article/148469/article.html.
6. Rumsfeld, D. 2002. Press conference (February 12); http://www.c-span.org/video/?168646-1/DefenseDepartmentBriefing102.
7. Stephenson, N. 1995. *The Diamond Age*. Bantam Spectra.
8. Stoll, C. 1989. *The Cuckoo's Egg*. Doubleday.
9. Tehranipoor, M., Koushanfar, F. 2010. A survey of hardware Trojan taxonomy and detection. IEEE; http://trust-hub.org/resources/36/download/trojansurvey.pdf.
10. Thompson, K. 1984. Reflections on trusting trust. *Communications of the ACM*, 27 (8); http://cm.bell-labs.com/who/ken/trust.html.
11. Venizia, P. 2008. Sorting out the facts in the Terry Childs case. *CIO*; http://www.cio.com.au/article/255165/sorting_facts_terry_childs_case/?pf=1.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

THOMAS WADLOW is a network and computer security consultant based in San Francisco. He enjoys the many cafes there, but never, ever uses his keys to hold an outdoor table.

acmqueue

Automated QA Testing at EA: Driven by Events

A discussion with Michael Donat, Jafar Husain, and Terry Coatta

To millions of game geeks, the position of QA (quality assurance) tester at Electronic Arts must seem like a dream job. But from the company's perspective, the overhead associated with QA can look downright frightening, particularly in an era of massively multiplayer games.

Hence the appeal of automated QA testing, which has the potential to be faster, more cost-effective, more efficient, and more scalable than manual testing. While automation cannot mimic everything human testers can do, it can be very useful for many types of basic testing. Still, it turns out the transition to automated testing is not nearly as straightforward as it might at first appear. Some of the thorniest challenges are considered here.

At EA, Michael Donat is an advocate of automation. His current focus is process improvement on the Player and Business Analysis team. He was previously the manager of QA at Silicon Chalk and ActiveState Corp., and has worked at Microsoft as a software design engineer.

Joining the discussion is Jafar Husain, a lead software developer for Netflix. Previously he worked at Microsoft, where one of his tasks involved creating the test environment for the Silverlight development platform. There he was introduced to MVVM (Model View ViewModel); he's a convert, he says, and now likes to spread the gospel of MVVM where applicable.

Terry Coatta, a member of the ACM Queue *board, brought this group together to discuss the potential for automated QA testing. He and Donat once worked together at Silicon Chalk, where creating a sophisticated test environment was among their challenges. Coatta is now the CTO of Marine Learning Systems, developing a learning management system for marine workers.*

TERRY COATTA In terms of your efforts so far to apply automated QA testing at EA, I gather you've found the going a little bumpy.

MICHAEL DONAT We started the journey thinking automation was a good idea, but then we tried it, and it failed. Still, we figured out what was wrong, and we fixed it. But, while we made it to a nice plateau, we realized there was still a long way to go. Our solution clearly wasn't going to get us everything we wanted—which was a way to broadly apply automated testing. To get there, and for some other reasons, several of us have concluded that what we really need is a new architecture along the lines of MVVM.

JAFAR HUSAIN What exactly was your driver for automating in the first place?

MD Our primary motivation had to do purely with the cost of manual testing, which has become quite significant given the complexity of our games. Basically, code changes that require us to retest everything manually can be incredibly expensive. By reducing those costs, we felt we would have an opportunity to redirect our testers away from what I call "stability testing"—which is something automation is capable of handling—so they can start focusing more on the authenticity and fun of our game experience.

TC In terms of stability testing, what did you see as your first opportunities for automation?

MD We started looking at this seriously when we were working on EA Sports' FIFA 10 [soccer game].

Initially, that involved 10 vs. 10 gameplay, which then became 11 vs. 11 with the addition of goalies. So we needed that many testers—either 20 or 22. But that's not all, since we also needed to test for interactions between different matches to make sure the server wasn't getting confused about what information to send to which match. So, in addition to the testers required to cover one match, we needed to have at least one other match in play at the same time—meaning we actually needed to have 40 or so testers involved at the same time.

Then, even after we'd managed to get everyone organized, we might end up running into some trivial bug just seconds into the match that would bring the whole thing down. Besides being wasteful, that was extremely frustrating for a lot of people who could have been doing something more productive during that time. All that came together to make a pretty strong argument for automation.

TC What were some of the problems you encountered as you worked toward that end?

MD First, setting up an OTP (online team play) match in FIFA 10 required the user to go through a few screens. There were 20 consoles and the script was time-based, meaning it sent commands to the consoles and then waited for some prescribed amount of time for all of them to get into the right state. Then it would send out the next batch of commands. The goal was to move the consoles in lockstep through a set of screen transitions in order to set things up for gameplay: participants chose which side they wanted to play, what jersey they wanted to wear, what position they wanted to play, and various other parameters. All those things needed to happen in concert just to keep the programming for the game as simple as possible.

At the time, our primitive test-automation system made navigating the front end problematic. Timing had to be just right, or tests would fail. As a result, I began advocating for a means of skipping the front end altogether, but I was forced to change my point of view. During manual testing of FIFA 10 OTP, a number of issues came up—so many, in fact, that the budget for manual testing had to be increased significantly. The question around the organization was, "How can we stop this from happening in the future?"

That led me to analyze roughly 300 crash bugs for which we had obtained data in the QA cycle. Part of my goal was to see whether there was any significant ROI to be realized by continuing to pursue automation. I found that slightly more than half of our crash bugs were actually coming up in those initial screen transitions. It turned out I'd been telling the games developers exactly the wrong thing. That is, we really did need to do testing on the front end, as well as on the back end. We couldn't make automation simpler just by getting rid of the front end.

TC That's interesting. It seems like all that's happening on the front end is that people are choosing things from menus, so how likely are you to find bugs there? In reality, what looks like a simple process of choosing menu items actually amounts to a distributed computation. You've got 20 different things going on, with input coming from all these different places, and now all of that has to be coordinated.

MD Exactly. It became clear we needed a different mechanism altogether. Just sending control inputs wasn't going to be enough. We needed the test program to be aware of where it was on a particular console and then be able to move that forward in an error-correctable way.

The guys who had originally put together the test-automation framework for FIFA had realized this would be necessary, but the capability for handling it had rotted over the years and didn't really exist by the time we were ready to tackle FIFA 11. So, one of the things we had to do was get the details we needed to see coming out of the UI so we'd be able to tell where things actually were.

JH I guess that instead of driving things from the view layer—that is, going through the controller and the views—you needed to bypass that view and go directly to the model itself.
MD Believe it or not, we were not at that stage yet. At that point, we were just happy to have scripts that were far more reliable, simply because they knew where they were in the state of the program.
TC That way, you could actually close the feedback loop. Before that, you would send a command and then have to wait and trust in God that something wasn't going to happen in the meantime, whereas now you don't need to have that trust since you can verify instead.
MD Right. We got to where we had more of a controlled state transition. Another big QA improvement we made on FIFA 11 was the addition of Auto Assist, whereby automation could be left to run the game itself while one or two manual testers drove the actual gameplay by supplying controller inputs for selected consoles. They didn't need to have 20 people on hand. That represented a huge improvement.
TC Some people might have just rested on their laurels at that point.
MD Maybe, but it was just one step for me. While introducing some test automation to specific applications like FIFA OTP is a wonderful thing, what I really want is a much broader application for stability purposes because that's what will make it possible for us to focus our testers on the overall game experience. That's the way to go about building a superior product.

The work on FIFA 11 helped convince EA of the potential benefits of automated testing, but accomplishing that end was clearly going to require a different architecture. The answer seemed to lie with the MVVM paradigm, an architectural pattern largely based on MVC. MVVM facilitates a clear separation between the development of the graphical user interface and the development of the back-end logic, meaning it should allow EA to separate OTP gameplay testing from UI testing.

TC Looking back on where things stood once you'd finished with FIFA 11 test automation, what did you see as your next steps?
MD As encouraging as FIFA 11 proved to be, the problem was that we had to spend a ton of time coding. Mostly that's because during game development, changes frequently would be made to certain screens, and then we would have to make corresponding changes in our test-automation script. That wasn't always properly coordinated, so things would end up breaking. As a result, we had a very fragile test-automation script that required a software engineer virtually dedicated to working on maintenance.

In the case of FIFA 11 OTP, that expense was justified, but I couldn't make the case for applying a similar level of test-automation effort across every other area of the game. We had to continue relying on a large number of manual testers to cover the full breadth of testing. Which made it pretty obvious we needed to figure out a way to encode our tests so that ongoing maintenance could be performed less often, using less expensive resources.
TC And that led you where exactly?
MD Basically, it meant the architecture would need to change. It should be easy to see how the game is laid out in terms of its screen transitions, but there should also be ready access to the data those screens act upon. In general, things should just be more accessible at a higher level of abstraction than is currently the case.
JH Is it fair to say you would like to focus on workflows independent of the actual UI controls?

MD That's absolutely right. Once that became clear, we realized we needed a different architecture—something more like MVVM. That isn't to say it has to be MVVM; it just needs to be something that can provide that sort of capability.

TC What is it about the MVVM paradigm that's important?

MD Essentially, it allows us to separate the data used by the screens from the screens themselves. We need that separation so automation systems can gain access to the things they need to tie into.

JH It might be useful to contrast the MVVM approach with other patterns many developers might be more familiar with—MVC, for example. In an MVC architecture, both the controller and the view know about each other and send messages to each other directly. In an MVVM architecture, instead of a controller, you have a view model, which is just that—a model of the view. The view model stores the state of the view, and the view object decides how the state of the view model ought to be presented.

Unlike in the MVC pattern, the view model has no direct knowledge of the view. Instead of sending messages to the view directly, the view model updates the view indirectly via the observer pattern. When the view model is changed, it broadcasts those changes, and the view responds by updating itself. The main advantage of this is that it's possible to test that the view models are in the correct state without even instantiating the view objects, which would add many asynchronous operations (usually related to rendering) that in turn would have to be coordinated.

Testing new models this way is easy since your models expose methods that can be directly invoked. Testing logic through the view layer is much more prone to error since it requires waiting for buttons to load and relies on the delivery of brittle messages such as simulated mouse clicks and key events.

Anyway, as you've moved beyond FIFA 11, what additional steps have you taken toward an MVVM sort of architecture?

MD I should point out that improved test automation is only one benefit of MVVM. Several other groups at EA are also moving this way for a variety of reasons. The steps we've taken so far have mostly been to make the separation of the data from the screens more apparent. Unfortunately, FIFA has so many screens that we can't just go in and rewrite everything. What we can do, however, is to work the new paradigm into new features.

JH It's interesting that, in the face of so many challenges, you've chosen to evolve your architecture in this stepwise manner toward MVVM. It seems you've found it easier just to add new events or extra components that follow this new pattern and then start using those as you can. I presume that at some point the plan is to make a more wholesale transition to MVVM—or something like it—as that opportunity presents itself.

MD That is the plan because it's the only way we can actually go about it. It's going to be a while before we can achieve the full breadth of automation I'm pushing for, but at least we're moving in the right direction.

Our next challenge is figuring out how to specify our tests, since we now have an architecture that lets us access that stuff. But we still don't know what those tests ought to look like, how they should be packaged, or how to contain the information such that it's easy to maintain and makes sense to the people who maintain it.

TC What's the pushback on arguments for an MVVM-like environment? Are people afraid the transition would be too hard?

MD There's no doubt it would be hard. What makes it worse is that the software engineers would have to make those changes in lieu of adding some new features—which can be a very difficult sacrifice to justify. I can't even say exactly how much they would be able to save as a consequence of automation. The truth is that they probably wouldn't save all that much since we're just talking about moving manual resources from one kind of testing to another.

TC Do you think it would actually be more expensive to build in MVVM? Or is this really more about resistance on the part of the software engineers to making any changes to the way they're accustomed to working?

MD That depends on the underlying code involved. Also, we sometimes make incremental changes to existing features. That is, we sometimes need to rewrite features because they need to evolve beyond the original design. If we're about to rewrite a feature anyway, that certainly presents an opportunity to take the newer approach.

On the other hand, if we're putting in a new twist for an existing game mode or adding a small feature to something that's already there, it would be very difficult to do that the new way while all that old stuff is still around. That would only make those incremental changes all the more expensive.

JH It seems that, in order to get to a place where you've actually got something useful, you're going to need to move an entire workflow to MVVM. I suppose that's going to be difficult to accomplish incrementally.

MD That's right.

JH We've run into this at Netflix. So I think you've touched on something that's worth pointing out—namely, that it's one thing to have two different but similar libraries in a code base, while it's quite another to have two different paradigms within the same code base. When you're in that situation, it can be very difficult for onboarding developers to figure out exactly what to do. Have you found this to be a stumbling block? And has that caused any friction?

MD Absolutely. There are many FIFA developers all over the world, so the idea of unifying all of them in support of moving in more of an MVVM direction is pretty hard to imagine.

JH I wonder if the current attitude of those developers toward MVVM reflects the fact that the benefits you're touting will only be realized downstream. Beyond that, though, are they also aware that MVVM might be a better architecture in general for development, quite apart from any testing benefits?

MD Actually, I've been really impressed with the software engineers I've worked with here. They all seem to know what the right thing to do is. But time is also an issue.

JH Is it fair to say the developers don't have any objection to MVVM, and might even be very much in favor of making the necessary changes to use MVVM?

MD Often, I'll be talking to a group of game developers about some idea and they'll say, "Oh yeah, we already know we should go that route," but when it comes to implementation, they aren't able to follow through because of time constraints.

JH In terms of how you move forward, I gather you still have some questions regarding the architecture and that you're also still trying to figure out what the API for your testers ought to look like.

MD That's right, although I'd put the emphasis on specification rather than API, because programming is expensive. We're trying to determine how we can specify these things in such a

way that they'll be understandable, maintainable, and robust in the face of change. That is, in its purest form, you'd like to run an OTP test where you have 22 consoles, with 11 being assigned to one team and the other 11 going to the other side, along with the ability to associate all appropriate parameters with each.

Then the question becomes: how can you specify that in such a way as to cover a broad range of tests? And that's, of course, because each time you run a test, you would like to be able to do different things. If you've got a multiple-match situation, for example, you might want to roll through all the different teams, stadiums, and jerseys so that over the course of many weeks of testing, you would wind up cycling through as many different combinations as possible—and all of that by essentially specifying only one test. That's our goal, anyway, but it's not entirely clear at this point how we're going to manage that.

JH There really are two questions here: (1) Is it possible? (2) Does it scale? There are also some more advanced approaches you could use to build asynchronous tests, but would those then be accessible to junior developers or test engineers?

MD Right. There's no point in doing this unless we can do it in a low-cost manner.

The transition to automated testing has a significant cost dimension when it comes to the use of human resources. First, software engineers accustomed to doing things one way must be convinced to change. They must learn a new paradigm, move from synchronous to asynchronous programming, and perhaps even learn a DSL (domain-specific language) for writing event-based tests.

Second, it's essential to strike the right balance between the work done by lower-cost QA testers and that which is reserved for higher-paid specialists. This means taking advantage of the asynchronous nature of the game by emphasizing declarative tests that are started and gated by events, while designing tests orchestrated to play off those events. This could allow for large numbers of inexpensive coders to write the declarative tests, while a much more select set of expensive coders are left to focus on the more sophisticated orchestration issues.

JH Have you explored different languages that might make it easier for lower-skilled developers to write event-based tests?

MD I've been considering the possibility of using a DSL. What worries me, however, is that there was a time when we had to encode game information in the test code, and I'm afraid we might end up going back to encoding information in some other type of code if we were to choose the wrong DSL.

One of the properties of the DSL we'd be looking to use is a container for the game information that needs to be transparent enough so people can easily access that information. It's important the information can be accessed using vocabulary that both the QA people and the game producers are familiar with.

JH Understood. The line between where a DSL begins and a library ends can be somewhat blurry. But a DSL can also be embedded as part of the general-purpose programming language you already use.

MD At the moment I don't think we're really going to be looking at any if-then-else loop coding. We're probably talking only about tests at the level of stimulus and response—that is, "When the program responds in this particular way, then provide this sort of stimulus."

TC Jafar, have you had any experience with DSLs at Netflix?

JH We're actually currently struggling to make a similar transition—not so much for testing but

more for finding a better way to coordinate the concurrency in our application. What we're using for that now is the Rx (Reactive Extensions) library. The interesting thing about this library is that it has actually been integrated into C#, meaning you can use it at a much higher level than if-then-else to coordinate asynchronous processes. There's also a JavaScript version of Reactive Extensions, which is what we're using now.

While this should make things very easy, in practice we're finding that it's a very new way of thinking for developers—particularly those who have come from a background of if-then-else, imperative, top-down programming. And this is despite the fact that the Rx abstraction is at a much higher level and is, in fact, quite declarative and obviously flexible, powerful, and capable enough to handle all sorts of complex asynchronous operations. It's not so much a matter of this new language being any more or any less difficult to work with; it's just that when you come from a synchronous way of thinking, making the transition to programming asynchronously can be very challenging.

Asynchronous programming requires a significant investment in terms of learning something new and a whole different way of thinking about your code. Which is to say I'm skeptical you'll manage to find a DSL out there that can transform a synchronous programmer into an asynchronous programmer in a few weeks, or even over the course of a product cycle.

MD That's my fear as well. There's going to be a need for people in the loop who are skilled in asynchronous programming. Whoever is coding up these exotic OTP tests where we have two or three matches going on at the same time is definitely going to need those skills.

But the open question for me is: How can you do that and still have the QA people specify most of the tests? It would be fantastic if we could just get to the point where 80 percent of the game code could be covered by tests written by the QA people. And then if the other 20 percent of the OTP tests had to be written by highly paid specialists, so be it. I would be cool with that just so long as we could get a large proportion of the code covered in a lower-cost manner.

JH Those specialized developers might be expensive, but if they're using the right set of tools or languages or frameworks or paradigms, then you have the potential to squeeze a lot more out of them. There's real value in identifying those individuals who are naturally inclined toward asynchronous programming and intensively training them. Beyond that, I think we're starting to see more frameworks and tools that have the potential to yield some tremendous savings once you start leveraging them such that those specialists can produce six or seven or even eight tests a day instead of just two.

TC Initially I got the sense, Michael, that you were hoping to find a DSL that would let you take better advantage of QA personnel by enabling them to execute a reasonably broad set of tests. Meanwhile, Jafar, it sounds like your experience so far is that the asynchronous stuff is sufficiently complex that the real win lies in finding those people who have some natural talent for it and then making them super-efficient.

How is this going to play out long-term? Is asynchronous programming just so difficult that it's always going to be the province of power people? Or is there anything to suggest this can be made more accessible to less-sophisticated programmers?

MD I think we're going to see a mix of the two. There's going to be some significant portion of any product that will remain fairly straightforward, where the coding is likely to be the kind that can be handled by lower-cost individuals once you've got the right framework in place. But that framework is going to need to be set up by someone who understands asynchrony and who has the training and

experience to deal with other reasonably complex requirements. There's definitely going to be a role for some highly trained and talented individuals, but you also want to make sure you can leverage those efforts to make their contributions broadly applicable.

JH I'm a little pessimistic about that. We recently were looking to build some asynchronous frameworks on the server at Netflix, and I think some of our developers started out with a similar attitude, based on the assumption that maybe 80 percent of our asynchronous problems could be easily solved with a few helper methods. The idea was to provide some simple APIs for a few of the more common concurrency patterns so junior developers would be able to tackle most of the asynchronous problems. We discovered that simple APIs solved only about 10-15 percent of our end-user cases—not 80 percent. That's because it was very easy to fall off a cliff, at which point it became necessary to revert to dealing with primitives such as semaphores or event subscriptions.

It turns out that even seemingly trivial async problems are actually quite complicated. For example, if you're making a remote request, it will invariably require some error handling like a retry. If two operations are executing concurrently, you'll need a way to specify different error-handling policies for each operation. What's more, each of these operations might be made up of several other sequential and concurrent operations. In reality, you need a compositional system to be able to express such rich semantics.

I admit it's possible that some simple helper APIs might prove more useful for building tests since the requirements are less stringent than for app development. So maybe you're right, Michael, to think you can mix low-skilled programmers with highly skilled ones. What exactly that mix looks like remains to be seen, though.

MD I couldn't agree more. I think that's the big question.

TC On a somewhat different note, my group has been developing for asynchronous environments, and finite-state machines have worked really well in that regard. We've found them to be a stunningly good way to capture information about events and transitions and stuff like that. So what are your thoughts about using state machines and some kind of language built around that? Are state machines simple enough for less-skilled developers, like QA people, to use them effectively?

MD I certainly think state machines describe the mathematics well enough. A transition effectively amounts to a stimulus-response pair. So, yes, you can describe what we're talking about as hierarchical-state machines. And yes, that's the perfect mathematical paradigm to use for discussing this. But you can't present that to low-cost personnel and expect them to be able to do anything with it. What you can do, though, is to use those same mathematics to create the tools and the machinery that drives all this stuff. In terms of what you put in front of the QA people, however, that can't be anything more than what they already recognize as stimuli to the responses they're looking for.

JH I completely agree. It's true that the primitives are simple enough that everyone can understand how to hook up to an event, set a variable, and then move from state to state. In practice, however, those simple primitives don't mean the overall program itself is going to be simple. In fact, it's going to be quite complex because there are so many different moving parts.

Another approach to asynchronous programming that's gaining a following in the JavaScript world is around an abstraction called Promises that's being integrated into common JS and F#. That gives you an async type that provides for composability while letting you build asynchronous programs in a stateless way. That might be the model you end up having to embrace—a declarative

stateless way of describing an asynchronous computation—since that's a level of abstraction that lower-skilled developers might be able to take advantage of.

TC Could you provide an example of that?

JH What it comes down to is that there's a new way of thinking about asynchronous programs. The move away from GOTOs to structured programs raised the level of abstraction. Today we build asynchronous programs with callbacks and state machines, and these programs suffer many of the same disadvantages of the old GOTO-based programs: logic tends to be fragmented into many different pieces. We can resolve this problem the same way we resolved it earlier—by raising the level of abstraction. Instead of using callbacks and state machines to build asynchronous programs, we can model them as sequences of data. An event, for example, can be seen as a sequence of data—one, in fact, that has no end. There's no way a mouse-move event is going to be able to say, "Hey, I'm done." It just goes on and on forever.

It's interesting to note we already have a means for modeling sequences in synchronous programming: the familiar iterator is a synchronous way of moving through a data structure, from left to right, simply by continuing to request the next item until the iterator finally reports there's no more data. Erik Meijer, when he was at Microsoft, turned the iterator pattern inside out and found the observer pattern fell out. In mathematical terms, the observer pattern is the dual of the iterator pattern. This is a very important unification since it means anything we can do to an iterator can also be done to observers such as event listeners.

The significance here is that we have several high-level languages for manipulating data structures that can be expressed as iterators. The most relevant example is SQL, which I would argue is a very successful high-level language because it allows developers to create complex queries that are both easy to understand and powerful to use. Now, based on the discovery that the observer and iterator patterns are dual, Erik has managed to build a framework that allows an SQL-like language to be used to create asynchronous programs.

The idea is that events and asynchronous requests for data are collections, just like arrays. The only difference is that asynchronous collections arrive over time. Most operations that can be performed on a collection in memory can also be performed on collections that arrive over time. Hence we find that a DSL originally built into C# to compose synchronous sequences of data can also be used to compose asynchronous sequences. The result is a high-level language for building asynchronous programs that has the expressiveness and readability of SQL.

MD That's a step in the right direction. I'm definitely going to look into this further.

JH We're using this technology on our Xbox platform right now. It seems to be just what you're looking for, Michael.

TC Can you describe how Erik Meijer's Reactive Extensions work applies in a QA environment? Say you've got a bunch of consoles you need to drive through some sequences so you can verify that certain things are happening in the game you're testing. Where does Rx fit into that? What would you be querying in that circumstance and how would you be able to turn that into a test result?

JH That's a great question, since some people have difficulty seeing the connection between querying a database and creating a test. A test in its own way is actually a query along the lines of: "Did this stream fire and did that stream fire before some particular event fired, which then led to some other thing happening?" That's really no different from querying a table to see whether a certain condition is true.

TC We would still need some mechanism to drive the system through different states. Perhaps Rx could even be used for that. At each stage the query is going to come back as either true or false. If it comes back false, then we'll know the test didn't pass since the sequence of events we had been expecting didn't match the query we issued.

JH Exactly right. But this can be partitioned into two steps. The first is the one Michael already mentioned: transitioning the system so as to make it more observable, and by and large that's simply a matter of adding events that fire when interesting things occur. The second step involves building queries over those events. Those queries would be very, very declarative—they wouldn't be state machines at all—so you would be able to confirm that certain conditions are met as you drive through the system.

TC It sounds like you're applying this approach to a product now. Has that experience proved to be positive? Are you finding that the Rx syntax or the query syntax is something non-experts might be able to use to capture information about the system?

JH Thus far, I don't think the syntax has really helped as much as I'd anticipated. The real challenge is in making the leap to thinking about events as collections. Most people have spent their careers thinking about events very mechanistically. Although thinking about events as collections might be conceptually simpler, it may also prove difficult to make the transition at the organizational level, if only because it's so hard to break bad old habits. My sense, however, is that if you can find some developers who are already inclined toward functional programming, then when you give them these powerful new tools for asynchronous programming, you're going to be able to realize the sorts of economies we're talking about.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org