The Apple goto fail vulnerability: lessons learned

David A. Wheeler

2021-01-16 (original 2014-11-23)

This paper identifies lessons that we *should* learn from the Apple "goto fail" vulnerability. It first starts with some background, discusses the misplaced blame on the goto statement, focuses on identifying what could have countered this, briefly discusses the Heartbleed countermeasures from my separate paper on Heartbleed, and ends with conclusions. Some of these points have been made elsewhere, but this tries to merge them together in one place. Others have not been noted elsewhere, to my knowledge. For example, gcc quietly dropped its support for detecting dead code that can lead to this vulnerability, and detecting duplicate lines in source code might also be an additional useful countermeasure.

This paper is part of the essay suite Learning from Disaster.

1. Background

On 2014-02-21 Apple released a security update for its implementation of SSL/TLS in many versions of its operating system. The vulnerability is formally named CVE-2014-1266, but informally it's often called the Apple "goto fail" vulnerability (or "goto fail goto fail" vulnerability).

The essence of the problem is straightforward. The code included these lines [Apple2014] in function SSLVerifySignedServerKeyExchange:

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

The problem was the second (duplicate) "goto fail". The indentation here is misleading; since there are no curly braces after the "if" statement, the second "goto fail" is *always* executed. In context, that meant that vital signature checking code was skipped, so both bad and good signatures would be accepted. The extraneous "goto" caused the function to return 0 ("no error") when the rest of the checking was skipped; as a result, invalid certificates were quietly accepted as valid.

Many technical details are provided by [Langley2014] (ImperialViolet), [Ducklin2014] (Sophos), and [Arthur2014]. Some people discuss it and also suggest countermeasures, including [Barr2014], [Baxter-Reynolds2014], [vanDeursen2014], and [Sethi2014]. The National Vulnerability Database (NVD) entry for CVE-2014-1266 explains where the problem occurred this way: "The SSLVerifySignedServerKeyExchange function in libsecurity_ssl/lib/sslKeyExchange.c in the Secure Transport feature in the Data Security component in Apple iOS 6.x before 6.1.6 and 7.x before 7.0.6, Apple TV 6.x before 6.0.2, and Apple OS X 10.9.x before 10.9.2 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 step or (2) omitting the signing step." [NVD2014]

Exactly when the vulnerability was added is not clear. The vulnerability was not in the final version of iOS 5 version 5.1.1, released in May 2012; it was present in iOS version 6, which was publicly released in September 2012. [Arthur2014]

I have not found any data on *why* the line was duplicated, probably a side-effect of Apple's notoriously secretive ways. It could have been by accident or on purpose, but most people (including me) think it is probably by accident. Baxter-Reynolds speculates that "Either the line has been duplicated by accident, there was a "if" check above it that was deleted but the developer left the "goto" in by accident, or there was a problem merging the code written by two developers. (Merging is a little complicated for non-programmers to understand. From time-to-time two developers will work on the same code in individual copies of the master code. When two copies of the same code are sent back to that master code, the source control system that manages everything has to make decisions about how to combine the two developers work. This process can go wrong.)" [Baxter-Reynolds2014]. A plausible cause of code duplication is a copy/paste error [Arthur2014]. Steven M. Bellovin stated that, "I don't think that Apple's goto fail was a deliberate attack... [but if it was] it very clearly was not the NSA or other high-end intelligence agency. As I noted, this is too visible and too clumsy." [Bellovin2014]. The widely-respected Bruce Schneier was more neutral about its being possibly malicious: "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." [Scheier20014]. A few do think it was more likely to be on purpose. The gotofail.com FAQ author states that, "It is hard for me to believe that the second "goto fail;" was inserted accidently given that there were no other changes within a few lines of it. In my opinion, the bug is too easy to exploit for it to have been an NSA plant. My speculation is that someone put it in on purpose so they (or their buddy) could sell it." [gotofail.com2014]. As [Arthur2014] notes, "By "sell" he's referring to the fact that you can now sell "zero-day exploits" to nation states and security companies - garnering up to half a million dollars or perhaps more"; but in the end, "Arguing for nefarious conspiracy is - well, not very much". Because this is the sort of error that *should* have been detected by methods such as basic negative testing, and this code is published, I think it's much more likely to have been accidental.

My guess is that it was accidental, the result of a merge that went wrong. Errors happen during software implementation. But whether it was intentional or on purpose, this vulnerability is easy to detect. The real question is why this mistake was not found before it was delivered to customers.

There is also little information on how the vulnerability was found. [Arthur2014] reports that it "seems to have been from a line-by-line review of code". The fixing process was bizarre as well. In particular, it was fixed first on iOS, but not simultaneously on MacOS, leaving all Mac users (and the Apple update process itself) vulnerable to a dangerous 0-day vulnerability for four days [Arthur2014].

The impact on users' systems was extremely serious, since this made it easy for attackers to do "man in the middle" attacks against any secured connection using SSL/TLS (including "https://" URLs). Knowledge of it quickly spilled into popular culture; as shown above, there was even a T-shirt made about it [DesignedbyJonathan2014].

The longer-term impact (once it was fixed) is harded to gauge, but I think it's fair to say that Apple's reputation for competently developing systems was damaged for a time. The problem wasn't that someone had made a mistake - everyone makes mistakes. The problem is that this particular vulnerability was easily detected by a variety of countermeasures, including automated testing. Apple's failure to detect this easily-detected vulnerability, and then putting this dangerous vulnerability into production, raised very serious questions in the press about the trustworthiness of Apple's software develeopment processes. For example, Matt Baxter-Reynolds' ZDNet article "Apple's 'goto fail' tells us nothing good about Cupertino's software

delivery process" said that "The fact that Apple's infamous SSL validation bug actually got out into the real world is pretty terrifying" and "The real fail in all this comes from the fact that the bug was not detected by any automated test suites... Competence at automated testing is an indicator of sophistication of the software development team and its sponsors... The problem is that somehow Apple was able to put that code into production. Why on earth was that not caught by an automated test suite?"

I previously wrote about how Heartbleed could have been countered ahead-of-time; here I apply the same kind of thinking to this vulnerability. The sad reality is that in 2014 *every* major SSL/TLS implementation had a major extremely-serious vulnerability revealed in it, including OpenSSL (Heartbleed), Apple's (goto fail goto fail), GnuTLS (goto fail), and Microsoft's. Clearly we need to do better! I think we should look at past problems, see what we can learn from them, and apply those lessons.

2. Misplaced blame: The goto statement

A few people blame the mere use of the "goto" statement (e.g., [Barr2014]). However, I think pinning blame on the goto statement is misplaced. As Matt Baxter-Reynolds explains, "The 'goto' statement is quite old-fashioned... The problem with 'goto' is that it allows developers to create 'flow' in code that is unnatural... and that's bad because developers should always strive to make code easy to read. For me, who cares whether 'goto' is used or not. There is nothing unclear about the code that contains the bug... 'goto' may be old-fashioned, but it is not inappropriate in this setting." [Baxter-Reynolds2014]. Similarly, Sethi found that when using other constructs "the code was not necessarily a lot easier to read. I don't believe that the use of 'goto' significantly contributed to this problem" [Sethi2014]. In particular, if you are using C, avoiding the goto statement is likely to lead to a lot of duplicated code that might not be properly maintained in parallel, leading to other problems.

This is even supported by a recent emperical study. An empirical study of goto in C code [Nagappan2015] found that in almost all cases C program developers, when they use goto at all, use it in reasonable ways.

Banning "goto" statements would not have really solved the problem, so let's look elsewhere for solutions.

3. What could have countered this?

Here are some ways to detect this problem *before* delivery. As with my Heartbleed paper, I note if the approaches use dynamic analysis (require execution of the program), static analysis (do not require execution of the program), or a hybrid.

However, I am not trying to create a complete list. My paper on Heartbleed focused on how to identify it, in part because many mechanisms used to find security vulerabilities did *not* work with Heartbleed. The Apple goto fail vulnerability is quite different. In this case, there are many mechanisms that *everyone* developing security-relevant code should have been doing that would have immediately caught this vulnerability. Thus, there is no need to list more sophisticated measures (such as formal methods). If an organization cannot be bothered to use simple mechanisms low-cost mechanisms to counter vulnerabilities, it will not apply more sophisticated mechanisms either.

3.1. Thorough negative testing in test cases (dynamic analysis)

Negative testing is creating tests that should cause failures (e.g., rejections) instead of successes. Thorough negative testing in test cases creates a set of tests that cover every type of input that *should* fail.

At the very least, *every* SSL/TLS implementer should include, in their regression test suites, a large set of certificates and protocol requests that should fail... and make sure they do. If your tests only show that good inputs produce good outputs, then those tests are mostly *useless* for security. You need to have *many* tests to show that data that should be rejected is actually rejected. I have previously noted that negative testing could also have detected Heartbleed.

In many ways the Apple "goto fail" vulnerability is *much* more embarrassing for Apple than Heartbleed was for OpenSSL. Heartbleed exploited an obscure functionality, and the vulnerability involved an overread (something that many tools cannot detect). In contrast, the "goto fail" vulnerability subverted the *entire purpose* of the library, in a remarkably trivial way. This vulnerability showed that Apple has an extraordinarily poor or non-existent security-relevant test suite.

[Arthur2014] interviewed a former programmer at Apple, who explained that, "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." Dogfooding is great for finding user interface problems, which is something Apple is known for. However, dogfooding is useless for security, because it does not introduce the negative testing necessary for security evaluation. The same former programmer noted that, "From a good software engineering standpoint, this type of issue should have been found."

Many people, including me, came to this conclusion independently. [Baxter-Reynolds2014] bluntly states that "The real fail in all this comes from the fact that the bug was not detected by any automated test suites."

Developers who use SSL/TLS libraries often don't test either, sadly [Georgiev2012].

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

3.2. Properly detect and check unreachable code, e.g., via warning flags (static analysis)

The extra "goto fail" caused a whole lot of code to become impossible to execute; code that cannot be executed is called "unreachable code". The term "dead code" is also sometimes used for this situation, but the term "dead code" is also used by some to describe other situations. For precision, we'll stick to the term "unreachable code".

[Arthur2014] confirms that if Apple had used a tool that did proper unreachable code detection, this problem would have been immediately detected.

Let's be clear: unreachable code is not, by itself, necessarily a security vulnerability. In particular, if you just thoughtlessly removed unreachable code, that does *not* make the software any more secure, because the code was never going to run the first place. Instead, you should run such tools and then try to determine *why* the code is unreachable. If the code is unreachable because of some error (as in this case), fix the error, don't remove the code!

Many compilers have warning flags to detect unreachable code. In general, developers should turn on as many warning flags as they can, including one to do unreachable code detection. Detecting unreachable code in absolutely all cases is undecidable (as noted by [vanDeursen2014]), but detecting simple cases like this is

easy and certain.

However, there's an ugly twist to this story involving gcc, one that I haven't see emphasized elsewhere. The gcc compiler, a widely used compiler, once reported unreachable code. However, years ago the gcc developers removed this capability because the code was unstable [Taylor2011]. Even worse, it still accepts a parameter intended for the purpose, namely -Wunreachable-code, without any notification that it no longer does anything. I have verified that gcc version 4.8.3 still quietly accepts this option flag, yet it does nothing and fails to even report a warning that it does nothing. As a result, many developers may be using gcc with this flag enabled, thinking that it will report dead code (as it once did), even though it will *not* do so.

The lesson that compiler makers *should* learn here is that if you *stop* supporting a warning flag, you should at *least* report a warning if your user uses it anyway. Frankly, I hope that compiler makers will *add* warnings, never *remove* them, for situations that sometimes indicate a serious problem.

The lessons that *developers* should learn is to use multiple analysis tools, and enable some of the same functionality even if it seems to be redundant. For example, I've verified that clang version 3.4.2 not only accepts the `-Wunreachable-code` option, but that it actually works correctly and *can* detect dead code caused by a goto. (The clang option `-Weverything` enables `-Wunreachable-code` and other similar options). That doesn't mean you need to use the clang compiler to generate the *final* executable code. Many people use gcc to generate final code (for example), and they can continue to do so while also using clang warning flags to help them find problems (or vice versa). The point is that you should use multiple tools to detect vulnerabilities, including multiple compilers's warning flags and other static analysis tools, because using multiple tools increases the likelihood of finding problems ahead-of-time.

3.3. Always use braces, at least if it's not the same line (static analysis)

The C programming language permits omitting braces in certain cases; their omission in this case led to a serious vulnerability. Other languages with C-like syntax permit the same thing, including C++, Java, and C#.

A coding style rule that tends to eliminate this problem is to always use braces with "if" branches, at least if the branches are not on the same line as the "if" statement. E.G., use this format:

```
if (condition) {
  ... stuff ...
}
```

Some people would also permit omitting braces if the whole construct is on one line. After all, a merge that accidentally loses or duplicates a line won't change it, and there's little risk of confusing its meaning:

```
if (condition) stuff;
```

The usual reason for this format is that it makes it easy to add new actions to occur in a condition, without accidentally having the later actions occur at all times.

In a stack exchange coding style discussion, User36294 has this prescient description on 2011-09-09 that described this very issue and a common cause of it: "I use the braces method - for all the reasons above plus one more. Code merges. It's been known to happen on projects I've worked on that single-statement ifs have been broken by automatic merges. The scary thing is the indentation looks right even though the code is wrong, so this type of bug is hard to spot." [StackExchange]

Many coding standards and style guides already require or recommend this. The Mozilla Coding Style guide requires this: "Always brace controlled statements, even a single-line consequent of an if else else. This is redundant typically, but it avoids dangling else bugs, so it's safer at scale than fine-tuning." The Recommended C Style and Coding Standards by Cannon et al recommend "fully bracketed" form (where *all* code is surrounded by braces) [Cannon]. Michael Barr notes that the Barr Group's "Embedded C Coding Standard" book rule 1.3.a has the same kind of requirement: "Braces shall always surround the blocks of code (a.k.a., compound statements), following if, else, switch, while, do, and for statements; single statements and empty statements following these keywords shall also always be surrounded by braces". In short, Barr states that "Too few programmers recognize that many bugs can be kept entirely out of a system simply by adopting (and rigorously enforcing) a coding standard that is designed to keep bugs out." [Barr2014]. Green's "How To Write Unmaintainable Code" similarly argues (in its usual backwards way) that if you want to make code *hard* to maintain, "Avoid {}. Never put in any { } surrounding your if/else blocks unless they are syntactically obligatory. If you have a deeply nested mixture of if/else statements and blocks, especially with misleading indentation, you can trip up even an expert maintenance programmer." [Green].

Some tools (mainly style checkers) can detect cases where braces are not used this way. Vera++ can detect unbraced control structures (this is rule T019, "Control structures should have complete curly-braced block of code"). The gcc flag -Wparentheses will warn of unbraced situations in certain cases, but not in general (it would not find this instance, for example).

Not everyone agrees with this style, e.g., the Linux coding style instead says "Do not unnecessarily use braces where a single statement will do" [Linux]. The Linux style guide does save some vertical space, but with a slightly increased risk of dangerous mistakes. In this case, I think the loss of a little vertical space is worth it.

3.4. Use coverage analysis (dynamic analysis)

Obvious questions when you test are "how good is my testing?" and "have I done enough?" A common way to answer these questions is to use coverage analysis; the most common measures are statement coverage and branch coverage. As noted by [vanDeursen2014], even basic statement coverage would have been enough to reveal that these lines were not tested... and any effort to track down *why* (or to try to create such tests) would reveal the extraneous goto.

If you do not have statement coverage, then you have statements in your code that are *never* executed by any test.

Requiring full statement coverage for security- or safety-critical code is not that unusual. A widely-used specification in the avionics world for software development is DO-178B and its successor DO-178C, titled "Software Considerations in Airborne Systems and Equipment Certification". It defines levels A through E, where level A is if failure is catastrophic, and level C is where the failure is major. In both version, statement coverage is required at level C or higher. (This is noted by IBM's description of of DO-178B and Validated Software Corporations' Avionics Validation and Certification FAQ).

Statement coverage is actually a relatively *weak* measure for the tests of critically-important software. Developers of critically-important software should be using at least one other coverage measure, at least branch coverage (which is also relatively weak). As explained in [Buechner2012], "even if 100% statement coverage is achieved, severe bugs still can be present. Therefore, please keep in mind what 100% statement coverage actually means: All statements were executed during the tests at least one time. Not more and not less. We can assume that 100% statement coverage is 'better' (i.e. results in a higher software quality, whatever this may be) than say 70% statement coverage, but the software quality achieved by 100% statement coverage is certainly not sufficient for a safety-critical project. If you [are considering applying]

statement coverage, [reaching] 100% is the minimal objective. Statement coverage can be useful for detecting 'dead code', i.e. code that cannot be executed at all. Statement coverage can [also] reveal missing test cases."

Some organizations will stop before getting 100% statement coverage and choose to manually examine the remaining code instead. That is a valid trade-off. In either case, all code is getting a basic examination, either by test or by review.

Frankly, I'm very tempted to add this to the "minimum set" - if you aren't measuring coverage in code that's highly security-sensitive, you have *no idea* how well you're testing in code that really matters.

You do not even have to achieve 100% coverage. In this case, simply measuring statement coverage and noticing when coverage went *down* unexpectedly would have probably detected this vulnerability.

3.5. Forbid misleading indentation (static analysis)

The vulnerability might pass a superficial code review because it had misleading indentation; one countermeasure is to forbid misleading indentation (through either reformatting or detection).

Arie van Deursen argues that "code formatting is a security feature" and that indentation "white space is a security concern. The correct indentation immediately shows something fishy is going on..." [vanDeursen2014]. Arie van Deursen also argues that code formatting should be done by tools, not by hand, and shows that clearly this code was not routinely formatted automatically. He notes that with Eclipse-format-on-save code could not be saved this way. However, van Deursen also admits that "hand-made layout can make a substantial difference in understandability of code. Perhaps current tools do not sufficiently acknowledge such needs, leading to under-use of today's formatting tools."

However, beware. Just reformatting on saving may have made this worse, unless it was followed by manual review. Reformatting code by itself would not have fixed it, since it would have had the same effect, and reformatting would have made it impossible for tools to detect the *inconsistent* indentation (which might have served as a useful warning). In addition, reformatting existing code creates merge difficulties (and thus can *contribute* to this kind of problem).

I think it is often a better approach is to use tools that report when indentation is blatantly inconsistent, or ones that allow more flexibility to retain hand formatting (and thus would be more acceptable by those who currently reformat by hand). Some style checkers can report inconsistent styles, which let people do manual formatting while warning of problems.

GCC (GNU Compiler Collection) version 6 adds a new warning for misleading indentation, `-Wmisleading-indentation`. This new warning immediately detects misleading indentation (such as the one in this code), and can immediately find them. This warning was added to GCC *after* this problem came to light (it was implemented in the 2016 timeframe), in part as a response to this specific vulnerability. If you use GCC version 6 or later, you should use this warning flag and fix problems it finds. Even better: this flag is enabled by default in GCC if you use `-Wall` ("enable all important warnings"), so if you simply use `-Wall` you'll get this check automaticlaly on newer verasions of GCC.

In 2019 (more recently) the clang compiler also added support for the `-Wmisleading-indentation` flag. However, this is *not* enabled by `-Wall` in clang (at least at the time of this writing), so you need to specially enable it if you only use clang and not GCC. Even before this option was added the LibreOffice developers were using a clang plug-in specifically to detect this kind of misleading indentation in their code, so that they can prevent these kinds of problems.

I've also confirmed that Kitware's KWStyle program can be configured to find this. KWStyle can report indentation that is inconsistent with the opening and closing braces, and that is enough to detect this particular problem. KWStyle is open source software released under the New/Revised 3-clause BSD license, and it existed when this vulnerability was reported.

Manual review preceded by reformatting (or at least preventing misleading indentation), would have almost certainly countered it as well. We will discuss this later in the section on manual review

3.6. Detect duplicate lines in source code (static analysis)

If merge errors can sometimes produce erroneous duplicate lines, then perhaps it'd be helpful to *check* for duplicate lines. Duplicate lines are not necessarily an error, but perhaps they are rare enough to specifically check.

To test this theory, I looked for duplicate lines in C code (.c and .h files) in the Linux kernel version 3.17.4. This version of the Linux kernel has 17,088,122 lines in .c and .h files including blank and comment lines. When looking for duplicate lines I used "uniq -d" and intentionally ignored blank lines, lines with just "*" (which are in comments and thus irrelevant), and "#endif" (the compiler detects unmatched endifs anyway).

In this test I found that there were 10,682 incidences of duplicate lines; this works out to one incident of duplicate lines for every 1,600 lines on average. That could be used, but it is a rather noisy measure. However, it turned out that nearly all of these incidences were in a few files that used code to represent data (primarily fonts and a logo file in arch/m68k/platform/68000/bootlogo-vz.h). It would be easy to look for duplicate lines and exclude the few files with many duplicates (to reduce the number of false positives). When I do that excluding "fonts" and "bootlogo", I get only 5,769 incidences of duplicate lines, or only one incident of duplicate lines for every 2,962 lines on average. That takes much less time to do, and it shouldn't take long to examine a duplicate line to determine if it suggests a more serious problem. Where duplicate lines are intentional, you could an on-line comment (of *any* kind) to note the duplication is intentional - then the lines would no longer be reported as duplicates.

Duplicate checking should not replace other mechanisms that can detect more problems, such as negative testing and enabling warnings (including searching for unreachable code). That said, this might be a useful additional measure to apply to software that is vitally important, especially if you know that your version control software (especially its merge routines) occasionally make this mistake. For example, you could apply the checking and *only* report duplicate lines in code involved in a merge as a potential warning for code to check. In this case, it would be trivial to do and could be automatically applied (e.g., in git this could be implemented as a client-side hook that warned if a merge produced duplicate lines that were not there before).

3.7. Change error-handling idiom or switch to a safer language to use exceptions (static analysis)

An indirect cause of the problem is that C does not have a full exception-handling mechanism (it has a mechanism called setjmp/longjump, but this is not the same thing).

This means that in practice, software developers using C will typically return error codes in their return values. This means that almost every call to a function must also handle error returns. The actual code [Apple2014] is a disturbingly long list of lines in the form (notice that *only* the call to do_something is the primary function; everything else is a wrapper to deal with errors:

```
 if ((err = do_something(...)) != 0)
   goto fail;
```

Arie van Deursen [vanDeursen2014] points out that this is an underlying cause of the Apple goto fail vulnerability. In short, the complexity caused by the "return error code" idiom is difficult to manage. As evidence, he points back to his earlier paper [Bruntink2006], where they analyzed a 15 year-old, real-time embedded system with approximately 10 million lines of C code. They found that there were 2.1 deviations from the return code idiom per 1000 lines of code. These deviations are not necessarily errors or vulnerabilities, but it does clearly show that developers often fail to follow the idioms. This shows that "the idiom is particularly error prone, due to the fact that it is omnipresent as well as highly tangled, and requires focused and well-thought programming."

The authors propose using specialized C macros that make the idiom much easier to apply. The specific macros they developed were as follows (I have tweaked NO_LOG so it uses explicit braces):

```
#define ROOT_LOG (error_value, error_var) \
  error_var = error_value; \
  LOG(error_value, OK);

#define LINK_LOG (function_call, error_value, error_var) \
  if (error_var == OK) { \
    int _internal_error_var = function_call; \
    if (_internal_error_var != OK) { \
      LOG( error_value, _internal_error_var); \
      error_var = error_value; \
      } \
  }

#define NO_LOG (function_call, error_var) \
  if (error_var == OK) { \
    error_var = function_call; \
  }
```

These simplify the number of control flows as seen by the programmer, which helps. However, as written these do not address resource release (an issue the original goto fail code had to deal with), so a variant of this approach may be needed in other systems.

It is possible to implement exception handling-like mechanisms in C [Paltanavicius2005], but as these are not well-supported it seems risky to depend on them.

Another approach, not specifically recommended by the authors, is to switch to a language that provides better mechanisms for error-handling (e.g., exception handling).

3.8. Make access failure the default (static analysis)

Real code has bugs. The "goto fail" bug turned in a serious security problem because the default value for the errorcode was 0 (no error).

What should have been done, instead, was to ensure that the default error code was an error (some nonzero value in this case), and then return no-error only after all tests had positively determined that things were fine. In this approach, skipping validation code would cause all validations to report an error. The bug

could have still occurred, but it would never have been a security problem. Instead, it would have been immediately detected by normal testing, and if testing was that bad, users would quickly report it in the field.

Walt Sellers eloquently noted this in an email to me on 2017-01-14: "In brief: an overlooked basic problem of the 'goto fail' code was the assumption of success until failure was proven. If failure was assumed until success was proven, the bug would not have existed. With this change in assumption, incorrect behavior of the code would tend to give incorrect denials of access rather than incorrect granting of access. Bugs would be noticed because people would complain that they were kept out. As you point out people do not complain of being let in."

3.9. Manual review (static analysis)

Just about any manual review is very likely to have found this. Simply having a duplicated goto statement is highly suspicious, *even if* the reviewer didn't immediately realize that the indentation was misleading. If the indentation had been cleaned up before review, just about any serious manual review would have found it. That's because with proper indentation this problem is very obvious to manual review.

As Sethi reports, "whenever I manually review code, I use the IDE to automatically fix indentation before I start looking at it. This doesn't guarantee that the reviewer will catch every problem, but it does make code easier to read and understand. If you perform manual code reviews, I would highly recommend doing this." [Sethi2014].

4. Heartbleed countermeasures

Heartbleed was a much more challenging vulnerability to find ahead-of-time. I have previously listed some of the mechanisms that could have found Heartbleed; here they are, along with notes on whether or not they would have countered Apple's goto fail:

1. Thorough negative testing in test cases (dynamic analysis): **Yes**. As I discussed earlier, thorough negative testing would certainly have found this vulnerability as well, and it should have been done.
2. Fuzzing with address checking and standard memory allocator (dynamic analysis): **No**. Fuzzing that focused just on crashes would not have helped, even with address checking, since this was not a memory access problem. See below for a variant of fuzzing that *would* have worked.
3. Compiling with address checking and standard memory allocator (hybrid analysis): **No**. Again, goto fail was not a memory access problem.
4. Focused manual spotcheck requiring validation of every field (static analysis): **Maybe**. Manual review of that particular set of code would have almost certainly detected this. However, in a focused manual spotcheck, what matters is the focus; if this part of the code was *not* reviewed, then the problem would not be found, and it is not clear that this part of the code would have been especially reviewed. If it only focused on initial validation, maybe, and maybe not.
5. Fuzzing with output examination (dynamic analysis): **Yes**. Just about any output examination would have determined that far too many certificates were being accepted, yet should have been rejected.
6. Context-configured source code weakness analyzers, including annotation systems (static analysis): Probably. This was a fundamental failure in the library's primary purpose, so a focused analysis is more likely to have worked.
7. Multi-implementation 100% branch coverage (hybrid analysis): **Yes**. Practically any test with bad certificates would have found this vulnerability. A multi-implementation branch coverage test would

have had to include such tests. Indeed, as noted earlier, using coverage analysis and requiring full statement coverage would have found this.

8. Aggressive run-time assertions (dynamic analysis): **Probably not**. In this case, many of the run-time assertions would probably have been skipped as well.
9. Safer language (static analysis): **Maybe**. However, as noted earlier, most languages with C-like syntax (including Java and C#) have similar problems. Many languages that are notably *not* C-like in syntax, such as Ada, do not have this problem because they do not allow the single statement branches that were at the root of this problem.
10. Complete static analyzer (static analysis): **Probably**.
11. Thorough human review / audit (static analysis): **Yes**.
12. Formal methods (static analysis): **Yes**.

However, as I noted earlier, there is no need to list more sophisticated measures (such as formal methods). If an organization cannot be bothered to use simple mechanisms to counter vulnerabilities, they will not do more either.

5. Conclusions

The Apple goto fail vulnerability was a dangerous vulnerability that *should* have been found by Apple. There are many mechanisms they could have used. Some include:

1. Thorough negative testing in test cases (dynamic analysis)
2. Properly detect and check unreachable code, e.g., via warning flags (static analysis)
3. Always use braces, at least if it's not the same line (static analysis)
4. Use coverage analysis (dynamic analysis)
5. Forbid misleading indentation (static analysis)
6. Detect duplicate lines in source code (static analysis)
7. Change error-handling idiom or switch to a safer language to use exceptions (static analysis)
8. Make access failure the default
9. Manual review (static analysis)

At the very least, I would recommend thorough negative testing in test cases, properly detect and check unreachable code, e.g., via warning flags, always use braces, at least if it's not the same line, and use coverage analysis for critically-important software like an SSL/TLS libary. These approaches would detect many other problems, and that array of techniques would have certainly countered this one. The other techiques listed above are promising also.

A related problem is that Apple is notoriously secretive. Apple has not provided more detail on exactly how the vulnerability got in, or why Apple failed to detect this problem (even though it was easy to detect). This secrecy makes it unnecessarily hard to learn from it.

The real problem, of course, is that lessons are often not learned. Apple seems to depend on using its own software internally as a quality assurance technique. That approach often works for finding user interface defects, but that approach is basically useless for security. This fact has been known for decades - though

perhaps not by Apple. I hope that Apple has (or will) change its software development processes so they will be more likely to detect vulnerabilities in their software *before* the software is deployed. It is not just Apple, either. I hope that *all* software developers will learn from the mistakes of the past, and then change how they develop software to prevent repeating those mistakes.

If you enjoyed this paper, you might also enjoy the entire suite of related papers in my essay suite Learning from Disaster. This includes my similar essays about Heartbleed, Shellshock, and the POODLE attack on SSLv3.

6. References

[Apple2014] Apple Inc. sslKeyExchange.c. http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c?txt

[Arthur2014] Arthur, Charles. Apple's SSL iPhone vulnerability: how did it happen, and what next? 2014-02-25. http://www.theguardian.com/technology/2014/feb/25/apples-ssl-iphone-vulnerability-how-did-it-happen-and-what-next.

[Barr2014] Barr, Michael. Apple's #gotofail SSL Security Bug was Easily Preventable 2014-03-03. http://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/

[Baxter-Reynolds2014]. Baxter-Reynolds, Matt. "Apple's "goto fail" tells us nothing good about Cupertino's software delivery process". 2014-03-20. http://digwww.com/news-10150464/apple-s-goto-fail-tells-us-nothing-good-about-cupertino-s-software-delivery-process.html.

[Bellovin2014] Bellovin, Steven M., Speculation about Goto Fail, 2014-02-24.

[Bruntink2006]. Discovering Faults in Idiom-Based Exception Handling. Bruntink, Magiel, Arie van Deursen, and Tom Tourwe. 2006. ICSE. http://www.st.ewi.tudelft.nl/~arie/papers/exceptions/icse2006.pdf

[Buechner2012] Buechner, Frank. "Is 100% Code Coverage Enough?" Hitex Development Tools Gmb. http://www.hitex.com/fileadmin/pdf/products/tessy/white-papers/WP-TESSY-0104-CC_e.pdf

[Cannon] Cannon, L.W., R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, Henry Spencer, David Keppel, and Mark Brader. *Recommended C Style and Coding Standards*. https://www.doc.ic.ac.uk/lab/cplus/cstyle.html

[DesignedbyJonathan2014] DesignedbyJonathan. "goto fail; T-shirt (plain code)". Artwork designed by DesignedbyJonathan. Made by Zazzle Apparel in San Jose, CA. Sold by Zazzle. Product ID: 235322391420181248. 2014-04-08 6:59AM. http://www.zazzle.com/goto_fail_t_shirt_plain_code-235322391420181248

[Ducklin]. Ducklin, Paul. "Anatomy of a 'goto fail' - Apple's SSL bug explained, plus an unofficial patch for OS X!" Sophos. 2014-02-24. https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/

[Georgiev2012]. Georgiev, Martin, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. "The most dangerous code in the world: validating SSL certificates in non-browser software". *ACM Conference on Computer and Communications Security*. 2012. pp. 38-49.

[gotofail.com2014]. Goto fail FAQ. https://gotofail.com/faq.html.

[Green]. Green, Roedy. "How to Write Unmaintainable Code: Ensure a job for life ;-)". Canadian Mind Products. https://www.thc.org/root/phun/unmaintain.html

[Langley2014] Langley, Adam. Apple's SSL/TLS bug. *ImperialViolet* (Adam Langley's blog). 2014-02-22. https://www.imperialviolet.org/2014/02/22/applebug.html.

[Linux]. Linux kernel developers. "Linux kernel coding style" *Linux kernel*. Version as of 2014-08-26. Note: Recent history of this document is available via git. Its first commit in the git repository is by Linus Torvalds on 2005-04-16 22:20:36, but this date is misleading, since this was merely the switch of the entire Linux kernel (version v2.6.12-rc2) to git. The original document was almost certainly by Linus Torvalds, but many others have edited it since. https://www.kernel.org/doc/Documentation/CodingStyle

[Nagappan2015] Nagappan, Meiyappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. An empirical study of goto in C code 2015-02-11. https://peerj.com/preprints/826v1/.

[NVD2014] National Vulnerability Database (NVD). CVE-2014-1266. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266.

[Paltanavicius2005], Paltanavicius, Adomas. Exceptions in C. 2005-04-10. http://www.adomas.org/excc/

[Schneier20014] Schneier, Bruce, "Was the iOS SSL Flaw Deliberate?", 2014-02-27.

[Sethi2014] Sethi, Amit. Understanding the Apple 'goto fail;' Vulnerability. Cigital. 2014-02-25. http://www.cigital.com/justice-league-blog/2014/02/25/understanding-apple-goto-fail-vulnerability-2/

[StackExchange] Stack Exchange. "Single statement if block - braces or no?" http://programmers.stackexchange.com/questions/16528/single-statement-if-block-braces-or-no

[Taylor2011] Taylor, Ian Lance (Google). Subject "Re: gcc -Wunreachable-code option". Mailing list gcc-help at gcc dot gnu dot org. 2011-05-25 06:21:52 -0700. Key text: "The -Wunreachable-code has been removed, because it was unstable: it relied on the optimizer, and so different versions of gcc would warn about different code. The compiler still accepts and ignores the command line option so that existing Makefiles are not broken. In some future release the option will be removed entirely." https://gcc.gnu.org/ml/gcc-help/2011-05/msg00360.html.

[vanDeursen2014] van Deursen, Arie. "Learning from Apple's #gotofail Security Bug" http://avandeursen.com/2014/02/22/gotofail-security/

Feel free to see my home page at https://dwheeler.com. You may also want to look at my paper Why OSS/FS? Look at the Numbers! and my book on how to develop secure programs.