Heartbleed

Anirban, ac4743@nau.edu

A report on the infamous vulnerability from 2014: https://heartbleed.com/

During the month of February in ‘14, Apple pushed a security update that affected the transport layer security protocol via SecureTransport for iOS (version < 7.0.6) and even OS X (version < 10.9.2) that basically allowed anyone on the internet who used that version of OpenSSL (which is a software library that secures communication over computer networks, and this was Apple's implementation of it) to access or read the memory of the servers that hold information for the web (all connections that went through that particular OpenSSL version). This potentially allowed attackers to eavesdrop on communications over websites or applications using that version of TLS or the OpenSSL protocol from Apple since the encryption could be bypassed.

This relates to the CIA classification triad in these ways:

1) Confidentiality is breached, as this bug not only allowed attackers unauthorized access to steal data on the internet through this protocol (for websites that followed that version of the protocol), but it also allowed them to impersonate others as there was no encryption scheme to check for a person’s digital identification, and one could possibly threaten or claim that he/she is an insider or employee for an organization as he/she had access to and retrieved org-specific data/secrets that became open during the lifetime of this bug’s existence.

2) Integrity might have been impacted in terms of trustworthiness for access to websites and applications on the internet that used Apple’s version of TLS for authentication and security. Consistency and accuracy of data shouldn’t be affected by this.

3) Availability is preserved in this case, as the information is in fact consistently and readily accessible for authorized parties (also available to unauthorized parties which is not desirable, but that does not fall under the definition of availability!).

In code, the vulnerability leverages an additional goto statement that comes after the second if-conditional in the function which verifies the exchange of signed messages (signed using a key given by some encryption scheme).

For reference, here is that function's code[1]: (source - ImperialViolet)

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen)
{
     OSStatus err;
     ...

     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;
     ...

fail:
     SSLFreeBuffer(&signedHashes);
     SSLFreeBuffer(&hashCtx);
     return err;
}
```

Now `goto` is in itself unconditional, but the intent of that particular `goto` was to be inside that if-conditional and to execute only when the `update` method for the SHA1 hash object does not return a value that would make `err` to be 0. The `goto` for the signed parameter check (again, the second `if` statement) would not be executed if the encryption is not met, which is what we would usually want. But since the second one is not inside the `if` statement's scope, it gets executed irrespective of the value of `err` (or even when the update method is not successful) and jumps into the `fail` label, which basically returns the status to be positive always, and the signature verification thus never fails. (meaning the encryption here is meaningless)

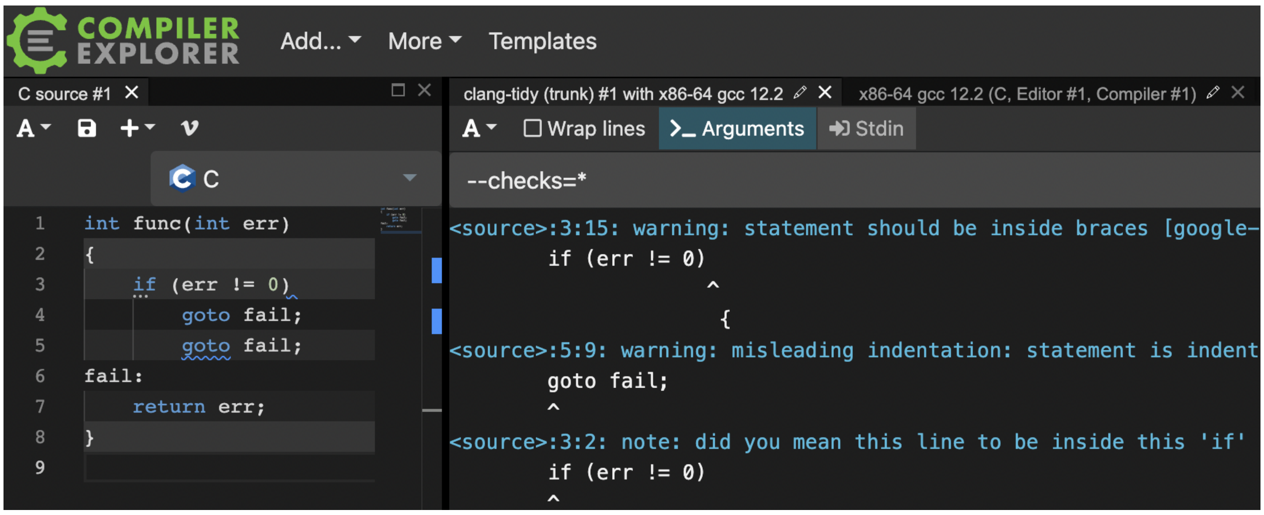
At present, I think that this sort of vulnerability can be detected through a static analysis tool in a fairly easy manner (given that it's just something out open in plain source code), either with a compile-time check which ascertains that goto statements do not lie out of a branch (which I think would be a custom rule, but then it's recommended to add your own rules to a static analysis tool for applying to one's use case software system, than to rely on the standard rules that apply to any project for a targeted language) or going the other way around, with a check that ascertains that if statements always use braces in order to precisely specify the statements that fall under the conditional.

The latter is established as a rule and comes to be detectable by most static analyzers. I decided to pick one named **Clang-Tidy** (alternatives for testing C code include PVS Studio, SonarQube, Coverity, Cppcheck, SVF, Infer, SourceMeter, etc.) to get this anomaly in source code discovered.
I've got the tool set up and integrated with Visual Studio Code locally, but since it has also been integrated with a prominent compiler explorer named Godbolt, I'll be sharing relevant screenshots from my use therein, and I choose this way since it can help the reader to test what I did himself/herself.

Running the static analysis tool using all modules and checks (i.e., with the flag `--checks=*`) on the full code produces too many results to examine (and some of them include stylistic changes which are not of our concern here; for e.g. `int func(...)` gets flagged due to it requiring a trailing return type check in Clang-Tidy from the `modernize-*` set of rules, which gets changed to `auto func -> (...) int` when using the flag `-fix`), so I'm going with a lucid[2], reproducible[3] example which can easily pinpoint the bug or core issue while avoiding all the noise surrounding it:

```
int func(int err)
{
	if (err != 0)
	    goto fail;
	    goto fail;
fail:
	return err;
}
```

Here's what the static analyzer reports for this:


COMPILER EXPLORER
Add...
More
Templates
C source #1
C
clang-tidy (trunk) #1 with x86-64 gcc 12.2
x86-64 gcc 12.2 (C, Editor #1, Compiler #1)
Wrap lines
Arguments
Stdin

```
int func(int err)
{
    if (err != 0)
        goto fail;
        goto fail;
fail:
    return err;
}
```

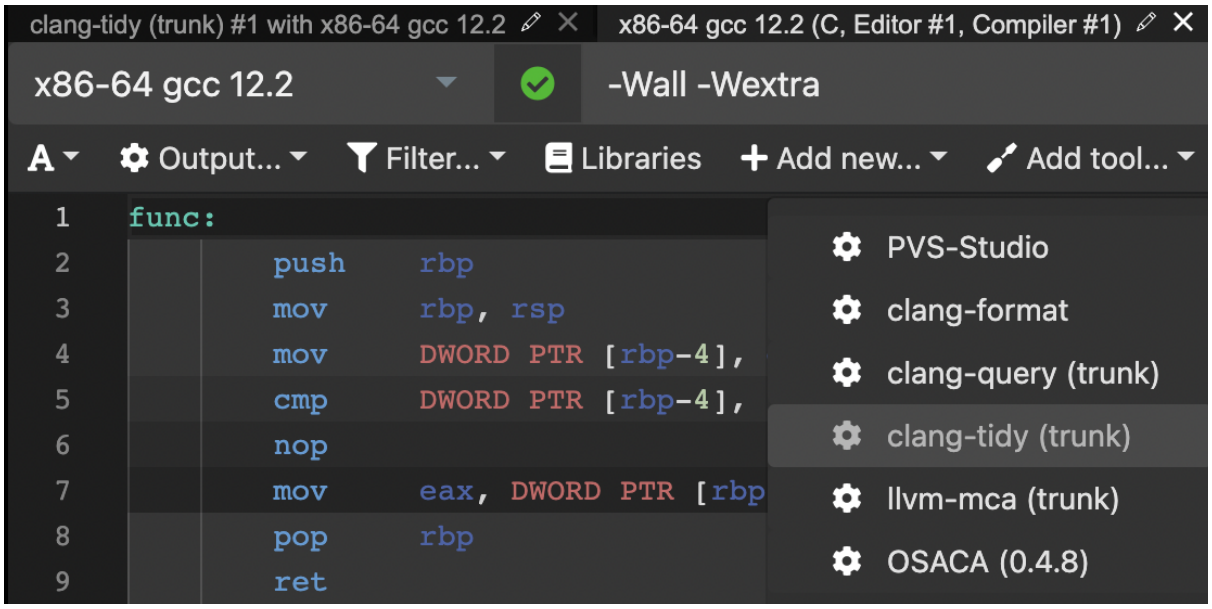
```
--checks=*
```

```
<source>:3:15: warning: statement should be inside braces [google-
        if (err != 0)
                     ^
                      {
<source>:5:9: warning: misleading indentation: statement is indent
        goto fail;
        ^
<source>:3:2: note: did you mean this line to be inside this 'if'
        if (err != 0)
        ^
```

As one can see, this illustrates the bug in its simplest form, i.e. the `goto fail` statements not being inside the scope of the `if` statement. Even if it were to be just one such statement under the if block instead of two, it's always better to have braces as that'll clear out these type of issues both when reading and writing code.

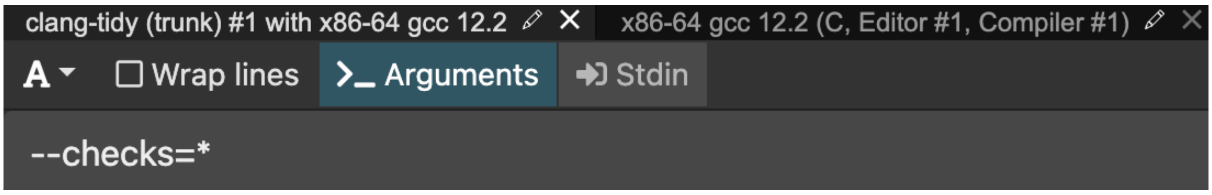
I drafted this example at the basic level for ease of understanding[2] (will move to other side problems with the code via an extended version below soon), to expose the problem within the code whilst keeping that bug logic consistent. And I say this is reproducible[3] since one can do this within a minute from a computer with internet access by following these steps: (i.e., you don't need to setup Clang-Tidy or even have a C compiler in your machine!)

1) Fire up a web browser, open up Godbolt.
2) Copy and paste my snippet above in the source code section to the left.
3) Select a C compiler, click/tap the 'Add tool' button and from the dropdown, select 'clang-tidy' and specify the arguments (again, I'm using all checks which is specified as `--checks=*` in the arguments field) - That's it!



x86-64 gcc 12.2 | -Wall -Wextra

```
func:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4],
        cmp     DWORD PTR [rbp-4],
        nop
        mov     eax, DWORD PTR [rbp
        pop     rbp
        ret
```


clang-tidy (trunk) #1 with x86-64 gcc 12.2
x86-64 gcc 12.2 (C, Editor #1, Compiler #1)
A
Wrap lines
Arguments
Stdin

```
--checks=*
```

Godbolt doesn't need you to be clicking anywhere to run either. As you type or modify your code, everything gets automatically compiled in the next moment and it goes along with your current state of code. Convenience at best, ain't it?

Moving on, it’s time to discuss about other real problems that were identified by Clang-Tidy near the vulnerable code. In order to inspect and possibly discover these ‘other problems’ (i.e. for code surrounding that bug) from this version of Apple’s OpenSSL implementation of that function, I extended my code above a bit:

```
#include <stdio.h>
int func(int statusVariable)
{
    int err;
    ...
    if ((err = statusVariable) != 0)
       goto fail;
       goto fail;

    (void)fprintf(stdout, "This program\
    never gets to reach this point.");
    ...

fail:
     ...
     return err;
}
```

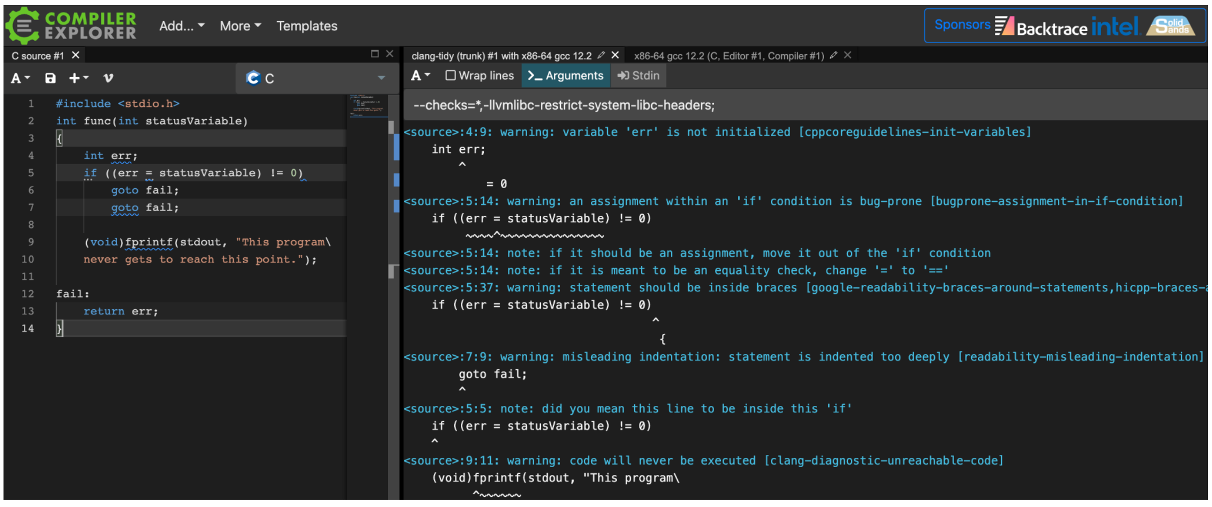
This modification makes the code identical to the structure of their original function[1], both of which have a few additional problems that the static analyzer clearly detects (well for the first one, I had to use the compiler flag `-Wunreachable-code` to make it appear) or hints about:

1) The first problem is the fact that there are certain statements or code in the program which are never executed since the logic as it is set up, would never reach there. For instance, consider the `fprintf()` call from my code above. The program would never reach that line during execution as that segment of code is skipped and the code following the label `fail` is executed. This happens because of the main problem which I discussed earlier, (i.e., due to the `goto fail` statements not being inside the scope of that `if` statement) making the condition redundant (it would jump to `fail`, either way), or making the call to that label a certain event for every program execution.

2) The second problem is the use of goto statements in a not very obvious and readable manner. Some static analyzers notify the user to avoid using goto statements, but to be fair they will work fine as long as they are used reasonably and in moderation (excessive use of them can lead to spaghetti code where it is difficult to trace the control flow of a program, making it hard to understand and modify). This blog post here talks about this (and also refers to Heartbleed) like a good debate for both sides of the argument.

3) The third problem is not really an issue and more of a false positive, but it has to do with good programming practices, like fixing some of the warnings that your compiler might have given. For instance, the variable `err` was only first initialized inside the conditional loop body, based on the passed parameter `statusVariable` in my code above (or from whatever the `update` method returned for the `SSLHashSHA1` object from the original code). Ideally, variables should be initialized to something aprior, and assignments should be done outside of `if` statements for the sake of clarity. It's not too difficult to make a mistake in such cases and later be perplexed about the wrongdoing when writing such snippets for a big program. Clang-Tidy too labels the assignment within an if condition to be 'bug prone'.

Here is what a run using the tool on Godbolt for my updated code looks like:



```c
#include <stdio.h>
int func(int statusVariable)
{
    int err;
    if ((err = statusVariable) != 0)
        goto fail;
        goto fail;

    (void)fprintf(stdout, "This program\
    never gets to reach this point.");

fail:
    return err;
}
```

```
--checks=*,-llvmlibc-restrict-system-libc-headers;
```

```
<source>:4:9: warning: variable 'err' is not initialized [cppcoreguidelines-init-variables]
    int err;
        ^
            = 0
<source>:5:14: warning: an assignment within an 'if' condition is bug-prone [bugprone-assignment-in-if-condition]
    if ((err = statusVariable) != 0)
         ~~~~^~~~~~~~~~~~~~~~
<source>:5:14: note: if it should be an assignment, move it out of the 'if' condition
<source>:5:14: note: if it is meant to be an equality check, change '=' to '=='
<source>:5:37: warning: statement should be inside braces [google-readability-braces-around-statements,hicpp-braces-a
    if ((err = statusVariable) != 0)
                                    ^
                                     {
<source>:7:9: warning: misleading indentation: statement is indented too deeply [readability-misleading-indentation]
        goto fail;
        ^
<source>:5:5: note: did you mean this line to be inside this 'if'
    if ((err = statusVariable) != 0)
    ^
<source>:9:11: warning: code will never be executed [clang-diagnostic-unreachable-code]
    (void)fprintf(stdout, "This program\
          ^~~~~~~
```

Note that I used the flag `-llvmlibc-restrict-system-headers` to filter out and discard the gratuitous warning for using `stdio.h` (or basically any C system header, which does not fall under compiler provided libc headers) since that is only applicable for the developers of the llvm-libc project. Anyways, another thing that is important while writing code is to maintain consistent indentation. As one can see above, Clang-Tidy too warns about the inconsistency in spacing for the second `goto fail`, flagging it as 'misleading indentation' (which in a way, gives away what's exactly wrong).