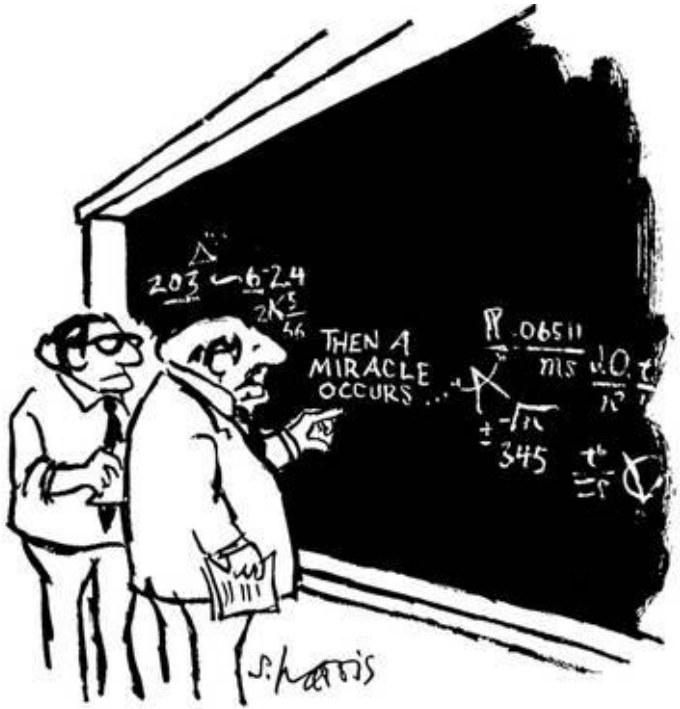
Do You Really Know Where Your Crypto is Executing?

Kelvin Desplanque
Compliance Engineer
11 May, 2018

“*All’s well that ends well*?”

- Cisco, like so many other organizations, has developed custom cryptographic libraries, based off of OpenSSL. Unfortunately, sometimes the use of our library, CiscoSSL and it FIPS Object Model (FOM), produces working cryptographic code which operates differently that was originally planned for.
- The objective of this presentation is not to show what didn’t compile or work.
- Instead, I will discuss what can happen when the code works perfectly fine (or at least appears to), yet was implemented in a fashion different to what was either intended or expected.

"*Then a miracle occurs …*"



- So how can this happen in the first place?
- How can it be detected?
- What can be done to correct it?
- How can future occurrences be prevented?
- … and yes … I will provide some real world examples.

The Root of the Problem

- Cisco has migrated from using home grown crypto to employing cryptographic libraries.
- We then migrated from fee-based licensed cryptographic libraries to open source-based cryptographic libraries.
- The move has been a good one and as most of you know the advantages are many:
  - Lower development and test costs
  - Lower or no licensing/royalty fees
  - Quicker responses and turnarounds to CVEs and other vulnerabilities.
  - Ease of use and deployment

There is no such thing as a free meal

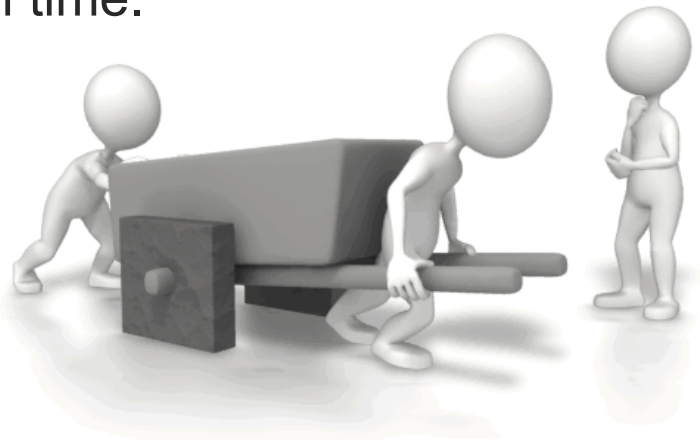
- Employing open source crypto does introduce some problems.
- In the past each BU (Business Unit) at Cisco would have a least one developer with a reasonably good understanding of basic cryptography.
- As a result of using common crypto libraries, much of this BU intelligence has waned. In many teams at best a small group of people (often only one person) understands the crypto library basic build requirements and functional call APIs.
- A minimal amount of effort is put into creating the make file so often mostly only the default options are selected when doing a build.

Meeting all the specs …

- The developer(s) responsible for implementing the crypto will make sure that all of the correct library calls are made so that the various cryptographic services will have the correct API calls to whatever cryptographic functions are required.
- The right functions will be invoked, with properly formatted and created keys, IVs, and whatever parameters are required.
- Test plans will be both created and executed to demonstrate that all cryptographic services work as per the requirements in the various specifications documents.

Meeting all the specs …

- FIPS mandated algorithm testing (CAVP) and functional testing (CMVP).
- The library (CiscoSSL) will have runtime tests, POST KATs, CRNGTs, etc. which confirm that the module is performing to both Cisco and FIPS 140-2 program requirements at run time.
- So if everything has been done perfectly, then how possibly could something be wrong?



The Role of the Compliance Engineer

- At Cisco, we have a number of people with the title Compliance Engineer.
- We wear many hats and our primary responsibility is to ensure that cryptographic modules can be successfully validated, whether is be FIPs, CC, Dod UCAPL, FedRAMP.
- We also have another role and that is to ensure that modules undergoing validation are constructed properly and will execute in the most secure fashion possible.
- We have to dig a little deeper to do this.

The Role of the Compliance Engineer

- In order to meet our mandate we must:
  - Develop a close relationship with the BU development team engineers.
  - Fully understand all of the functional requirements involving cryptographic functionality in the module.
  - Be acquainted with the basic operations of the cryptographic libraries.
  - Reach out to the engineering teams, which are responsible for designing and supporting the cryptographic libraries, as needed for both advice and support.
  - Be suspicious of everything and not be afraid to ask lots of detailed questions.

How to be vigilant

- Develop source code review processes which go above and beyond those that are required by the FIPs certification labs.
- When reading through design documents (system, software, hardware) try to make a point of zeroing in on words, phrases, illustrations, which may indicate special forms of cryptographic data processing or manipulation.
- As you find errors in implementation, make a note of these and add them to a "*lessons learned*" file which can then be used to perform investigations of future analyses.

So let us consider a simple example ...

- A software development team claims that they have moved away from target O/S's default entropy gathering mechanism and are now using a hardware-based entropy.
- Ask the developer(s) to urce code where they have provided a call in their n:
  `FIPS_drbg_set_cal`
- Let them walk you throu ne **`get_entropy`** input parameter to see if thei atever API exists for accessing the hardware

Another example ...

- A software development team has a custom Linux-based OS and has included **`rng-tools`** in the kernel. They claim that their entropy is purely S/W-based since they are invoking **`/dev/random`** to produce their entropy material. They run the NIST entropy estimation suite and provide you with the min-entropy calculations.
- You happen to note that the min-entropy value just seems to be too good for a purely S/W-based entropy source.
- Upon inspecting the source code and build files, you note the presence of **`rng-tools`** …. but you remember that this finds H/W entropy sources and you have an Intel Ivy Bridge CPU which has both ***`rdseed`*** and ***`rdrand`***.

And one last example ...

- A software development team claims that all of their crypto functionality is purely S/W-based since they only make crypto calls to a OpenSSL-based crypto library.
- When studying the H/W Functional Specification document you detect a Cavium CPU in the data plane. You know that this CPU was designed primarily for high speed IPsec acceleration.
- After you bringing this point up with the developers, they conveniently forgot to mention that this module intercepts both ingress and egress IPsec packets and redirects them to the Cavium CPU for both encryption and decryption, thereby bypassing the S/W crypto implementations in the crypto library.

Build configuration options ...

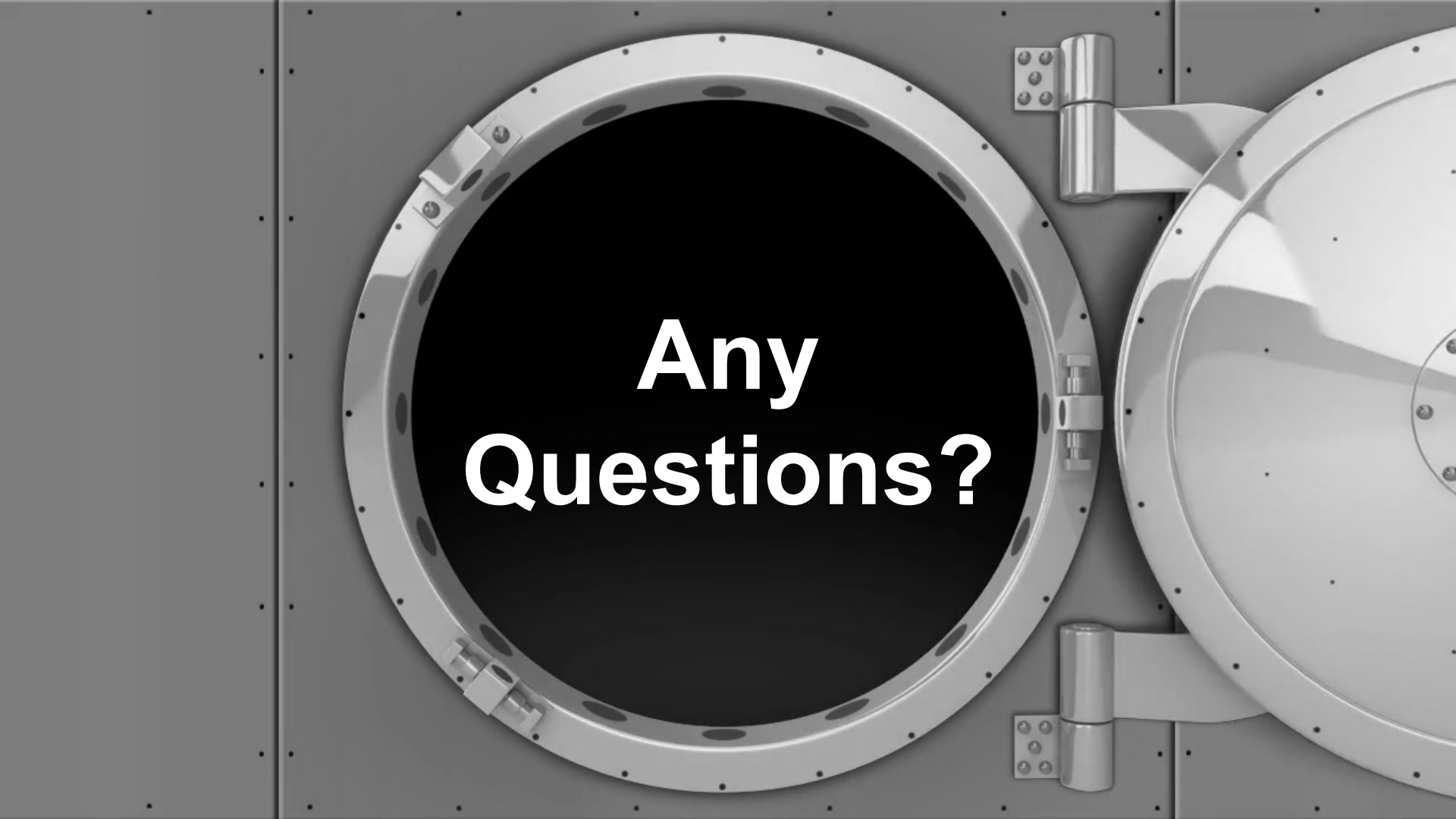
- There are a number of compile time options which can further easily reduce performance in a image:
  1) **`no-asm`** - Disables assembly language routines
  2) **`no-hw`**, **`no-engine`** - Disables hardware support

And finally, from our CiscoSSL dev team ...

- Environment not suitable to support load time POST, i.e. does not support constructors that can run in pre-main.
- Misunderstanding how and when to enter FIPS mode.
- Not understanding which processes and/or threads use the crypto.
- Not linking with the correct version of the crypto, e.g. many Linux environments have multiple versions of OpenSSL, does your application link and use the correct one?

So how can this help in future evaluations?

- Incorporate direct questions into future module Gap Analyses.
- Bulk up your S/W code reviews to looks for hints of problems and/or code snippets that can be used to back-up claims of properly written code and make files..
- Add additional text to the crypto library implementation and user guide documents.
- Suggest to development teams that their own internal Q/A processes incorporate some of the same material as described above.


Any
Questions?


CISCO