Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Pentest-Report SecurityDriven.Inferno 09.2016

Cure53, Dr.-Ing. Mario Heiderich, Jann Horn

Index

Introduction

"While many developers are aware enough not to roll their own crypto, they either pick the wrong approach, screw up the implementation, or both. I've written the SecurityDriven.NET book to highlight many challenges, misperceptions, and false assumptions of producing secure, implementationally correct .NET solutions. However, while recognizing the pitfalls of .NET cryptography is certainly useful, most of you would feel a lot more comfortable using an existing .NET library for common crypto needs rather than creating a risky ad hoc implementation. I know I would. Unfortunately, most .NET crypto libraries are awful. Many of these libraries focus on providing as many crypto primitives as possible, which is a huge disservice.

For example, if you follow "Internet advice", you are likely to come across the Bouncy Castle c# library (a typical StackOverflow recommendation). Bouncy Castle c# is a huge (145k LOC), poorly-performing museum catalogue of crypto (some of it ancient), with old Java implementations ported to equally-old .NET (2.0?). If you have a crypto archaeology itch, Bouncy Castle will scratch it. However, for typical practical purposes a new, modern, trusted, general-purpose .NET crypto library is required."

From http://securitydriven.net/inferno/

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

This report documents a penetration test and code audit of the SecurityDriven.Inferno library. The assessment was performed by two members of the Cure53 team in the second half of September 2016 and yielded only seven rather low-risk findings.

As for the test approach, the investigated library is available as an open source. Therefore, the audited code was taken from the public Github repository of the product, with the details listed below under "*Scope*". Since SecurityDriven.Inferno boasts a small size and compact design, the entirety of the code has been put in scope by the library's maintainer and received a complete coverage during this two-day assessment. The tests proceeded smoothly and the communication between the Cure53 team and the SecurityDriven.Inferno maintainer was fast and fruitful, leading to the reported issues being fixed quickly and in an appropriate manner.

Shedding light on the severity of the seven reported findings, it has to be noted that only three were classified as actual security vulnerabilities and the remaining four constituted general weaknesses. One of the spotted issues was considered to be of "Critical" impact and was immediately addressed by the maintainer. Nevertheless, the majority of issues should be seen as minor flaws and mishaps. For the sake of completion, it can also be added that the library maintainer discovered another vulnerability during the testing period, yet that issue is not listed in this report.

Note: All issues described in this audit report have been fixed by the library's maintainer. All fixes have been confirmed to be valid by Cure53.

Scope

- **SecurityDriven.Inferno Source Code**
    - https://github.com/sdrapkin/SecurityDriven.Inferno

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *SDI-01-001*) for the purpose of facilitating any future follow-up correspondence.

SDI-01-002 CryptoRandom.NextDouble() uses 32 bits of entropy *(Low)*

CryptoRandom.NextDouble() only uses 32 bits of entropy; a *double* could store 52 bits in the fraction component.

Although most callers requiring randomness for cryptographic purposes will probably not use *NextDouble()* anyway, it is recommended to document how much randomness the returned values contain.

SDI-01-003 Random Number Reuse through Thread-Unsafeness (*Critical*)

CryptoRandom implements a *NextBytesInternal* method which attempts to fulfill requests for small amounts of random data (<64 bytes) using a page-sized buffer *_byteCache.* That buffer is shared between threads. Likely for performance reasons, this code attempts to only use a single atomic operation (and its implied memory barrier) as a synchronization mechanism in the fast path. In that path, the request can be fulfilled with the use of the buffered-unused data. However, this method is not actually thread-safe.

Consider the following scenario:

1. The initial value of *_byteCachePosition* is 4088, so eight bytes of random data are remaining.
2. Both Thread A and Thread B concurrently request eight bytes of random data each.
3. Thread A starts running first. It performs the atomic add-and-return operation (*Interlocked.Add()*), verifies that there is sufficient data remaining in the buffer and enters the fast path.
4. Now Thread B runs. It also performs the atomic add-and-return operation, detects that the buffer does not have sufficient random data left. Therefore, it takes a lock on *_byteCache*, refills the buffer with new random data, copies some of the new random data to the output buffer and resets *_byteCachePosition.* This is so that future requests are able to use the cached data.
5. Thread B returns.

6. Now, Thread A continues running. It copies data from the end of *_byteCache* into the output buffer and returns.

The problem here is stems from the fact that Thread A reserves memory in Step 3, yet that very memory reservation is invalidated by Thread B in Step 4. There the memory is refilled and the *_byteCachePosition* is reset, so that in Step 6 Thread A reads memory that has not been reserved. Once both Thread A and Thread B have returned, *_byteCachePosition* is eight, which implies that only the first eight bytes of *_byteCache* have been used so far. However, actually the last eight bytes have also been used already by Thread A. As a result, a reuse of random data will take place.

It is recommended to never use low-level primitives like interlocked operations for synchronization in a security-critical code. High-level mechanisms provided by the standard library should be employed instead. In this case, static thread-local buffers could be used to implement a fast random number generator that does not require synchronization.

The same issue is present in *GetRandomUInt()* and *GetRandomULong()*.

SDI-01-006 Various Integer Overflows *(Low)*

Only few functions in the Inferno library perform explicit overflow checks when adding or multiplying length values and other integers. Most of the overflows will always cause a negative number to be used as an array index, causing an *IndexOutOfRangeException*. While the latter is not a very clean and optimal approach, it should not be a severe issue.

However, in the following cases the resulting behavior might be dangerous:

- In *EtM_Transforms.cs* it is not verified that the *currentChunkNumber* counter does not wrap. Theoretically, if around 360 Terabytes are streamed through *EtM_Transforms.cs* in the same session, this could permit chunk reordering attacks.

In the following cases, the function will eventually abort, but only after incorrect computations have been already performed. Once again this should continue to be seen as safe but might become unsafe after minor code changes, namely in the case outlined below.

- Both *Base32Extensions.ToBase32()* implementations compute *bitLength = length * 8* and allocate an output buffer based on *bitLength*. An overflow will cause the output buffer to be too small, which will exclusively be detected in the loop that writes into the output buffer.

It is recommended to, at the very least in the presented cases, use checked additions. Moreover it is recommended to investigate whether the performance impact of checked arithmetics is sufficiently low to feasibly employ checked arithmetics for all code, or at least majority of the code, with opt-outs for safe, hot code.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid attackers in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

SDI-01-001 Unchecked arithmetic in ConstantTimeEqual *(Low)*

The implementations of *ConstantTimeEqual* operate with five parameters shown below.

```
if (xOffset + length > x.Length)
        throw new ArgumentException("xOffset + length > x.Length");
if (yOffset + length > y.Length)
        throw new ArgumentException("yOffset + length > y.Length");
```

Both additions could overflow, thus bypassing the check. However, this would simply lead to another exception. The latter exception would occur when an attempt is made to access one of the arrays / strings out of bounds or at a negative index. It is recommended to either add the "*checked*" keyword for these checks or remove them.

SDI-01-004 AesCtrCryptoTransform can be misused *(Low)*

AesCtrCryptoTransform assumes that the caller is aware of the convention that *TransformBlock()* must only be called with full blocks while *TransformFinalBlock()* may be called with an input size that is not divisible by *AES_BLOCK_SIZE*. However, if a caller is not aware of the convention and misuses *TransformBlock()*, this could cause key-stream reuse because the counter is not incremented after processing a partial block.

It is recommended to let *TransformBlock()* verify that *inputCount % AES_BLOCK_SIZE* is zero. *TransformBlock()* and *TransformFinalBlock()* could then both call the same function, so that the actual task is completed this way.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

SDI-01-005 No error checking in FromBase16() and FromBase32() (*Info*)

The functions *FromBase16()* and *FromBase32()* fail to throw errors in the following cases:

- For both functions: when the input string contains characters that are not in the defined alphabet, the characters from outside the alphabet will silently be treated as characters with a value "0".
- *FromBase16()* only: *str16* has an uneven length - in this case, the last character will be silently ignored.

Despite this being rather unlikely to have security impact, it might still make sense to add the appropriate checks here.

SDI-01-007 Inappropriate constant name: KEY_LENGTH (*Info*)

The constant *KEY_LENGTH* in *SP800_108_Ctr.cs* is named inappropriately. More specifically, it is the length of the length of the key, not the length of the key. It is recommended to rename it to *KEY_LENGTH_LENGTH* or similar.

Conclusion

This two-day assignment, carried out by two testers of the Cure53 team in late September of 2016, revealed the tested SecurityDriven.Inferno library to be largely in line with what it promises to its users and quite the security-centered.

The aim of the project was clear and the scope reflected its purpose accordingly. The code was evaluated as cleanly-written and easy to audit, while the seven reported findings posed almost no major risks for the integrity and production-readiness of the SecurityDriven.Inferno library. Judging by the professional attitude of the maintainer, the Cure53 testers have no doubts that the issues get addressed quickly and the project moves forward on the right path once in production.

In sum, the library makes a positive and robust impression. In spite of one "Critical" finding, once all findings are addressed and fixed appropriately, the project can be considered production-ready.

Cure53 would like to thank Stan Drapkin as well as Chad Hurley of the OTF for their excellent project coordination, support and assistance, both before and during this assignment.