DCrypt - Block Device Cryptography

Max Zinkus - mzinkus@calpoly.edu

Project Motivation and Background

Initially, I wanted to implement work attacking nonce reuse on GCM, but I became very much interested in lab 4 and expanded the work out into a full C implementation of the following program. I ran nonce-reuse attacks against the GCM use within my implementation, and validated that the generation, incrementation, and authentication of nonces used in my code does in fact avoid nonce reuse up to a very large number of block encryptions, after which the recommendation is to change the data encryption key.

My motivation into delving further into this particular project was twofold, first, that the current implementations for block device encryption lack cryptographic guarantees about integrity, namely AES in XTS mode, which simply makes integrity violations very likely to cause decryptions to come out as random garbage data, and second, that XTS utilizes deterministic encryption which is also vulnerable to attacks on confidentiality.

It is also a prime time to move forward with a system with the security requirements that the header data has, as systems which utilize PCIe device isolation for security are becoming more popular and more usable. The specific example which comes to mind is Qubes OS, which runs under the Xen hypervisor and utilizes Intel VT-d to isolate the PCI subsystem, which could protect the header USB from unauthorized writes as described in the requirements below.

Construction Summary and Requirements

DCrypt uses a header, which is protected from untrusted writes during use, and a block device abstraction.

The contribution of this implementation is **authenticated, randomized encryption of a block device** in exchange for additional security requirements regarding the operational security of a small header. For a 16 terabyte drive, this header will be very slightly larger than 112 gigabytes. The intended use of this system involves use of a 128 gigabyte USB drive for storage of the header which is plugged in at device mount before password entry, protected by the operating system kernel from untrusted writes by unauthorized processes (a technology such as Intel VT-d or AMD IOMMU would likely be advantageous in preventing the violation of this requirement), and unplugged after unmount. Thus, the security requirements for this system have slightly larger scope than that of a usual cryptographic construction in that this system requires some operational, system-level security.

All other security assumptions are those of the confidentiality and/or integrity assumptions of the underlying AES-GCM, XSalsa20, and SHA-512 algorithms/constructions, and therefore the validity of those assumptions will go unquestioned by this analysis.

DCrypt Header

The header is laid out as follows:

```
Nonce | Salt | encrypted DEK | SHA-512 HMAC | GCM (nonce, tag) pairs
```

The header begins with a 192-bit XSalsa20 nonce, which is used for encryption of the data encryption key (DEK) with XSalsa20 from Bernstein.

XSalsa20 is a Salsa20 variant which uses an extended nonce. This nonce is sufficiently long that it does not negatively affect security to randomize it every time the data is re-encrypted. This allows us to maintain less state, and re-randomize the nonce at unmount-time.

The salt is used with the scrypt algorithm for key derivation. This allows us to derive two secrets from the user-supplied password, one of which is used as an XSalsa20 encryption key and the other of which is used to authenticate.

The user-supplied password is derived into a secret twice as long as needed for a secure encryption or authentication key, per the generally accepted method of deriving multiple secrets from a key using a PBKDF such as scrypt.

The key encryption key (KEK) derived from the user-supplied password encrypts the DEK, which encrypts and authenticates the data. Therefore the confidentiality of the DEK depends on the confidentiality of the KEK. Thus, the confidentiality and integrity of the disk depend indirectly on the confidentiality of the KEK, as well as directly upon the confidentiality and integrity of AES-GCM, and indirectly upon the confidentiality of XSalsa20 and the integrity of a SHA-512 HMAC.

The SHA-512 HMAC is used to authenticate the entire header at mount-time. The secret for this HMAC is derived using scrypt and the user-supplied password. This allows us to be sure that blocks were not re-arranged. The length-extension-mitigating construction for HMACs is used to avoid traditional length extension attacks against the header. Authenticating the header proves the sanctity of the XSalsa20 nonce and the GCM nonces in the `(nonce, tag)` pairs, preventing a nonce reuse vulnerability which in the case of GCM would allow forgery of device blocks.

Block Device

Using AES-GCM, the block device is encrypted and authenticated with the DEK, which is decrypted into memory at mount and encrypted into the header at unmount. Therefore the confidentiality and integrity of the blocks depends on a long, high-entropy key and the guarantees of AES-GCM.

The block device abstraction is transparently decrypted and encrypted on reads and writes, respectively. It is additionally authenticated based on the GCM tags provided in the header. The GCM encryption is also randomized using the nonces provided in the header. As such, assuming the nonces are not reused and the tags are sufficiently long, each block on the device reaps all the benefits of authenticated encryption, and assuming the SHA-512 HMAC in the header is not forged, the block device as a whole is vulnerable to the existential forgery of re-arranging validly authenticated blocks and their associated data.

Operational Security Requirement

As mentioned, the header data is intended to live resident on something like a 128 gigabyte USB to be protected by the user against tampering, and also as mentioned, between mount and unmount must be protected from unauthorized writes. Such unauthorized writes would evade mount-time authentication of the header, they could make a user re-use nonces in GCM encryption, which would allow for block forgeries utilizing the attacks presented by Devlin et al last year (2016).

Implementation Notes

I used libsodium, a C interface implementing Bernsteins library, NaCl, to access the various cryptographic primitives mentioned above. The entire system is written in C conforming to the C99 specification.

Methodology

My approach to developing a novel cryptographic construction was primarily to ensure that nothing aside from data indistinguishable from randomness was left on the block device. Secondarily, my goal was to mitigate the known attacks against AES-GCM, which is used for authenticated encryption of the disk. This meant ensuring the sanctity of the nonces to prevent block forgery. Thus, I thought that it would be critical to authenticate the `(nonce, tag)` pairs, and using an HMAC allowed me to also include the DEK encryption nonce and encrypted DEK in the authenticated data.

Modulo changes in the exact byte sizes of the nonces, tags, and precisely which algorithms are called, my implementation is relatively future proof to the possibility that the primitives it uses are compromised. As long as replacement cryptosystems offer the same cryptographic guarantees, they could be swapped into my implementation relatively simply, due to a focus on modularity in the code.

Future Work

Future work for this implementation project could delve into the performance aspects of doubling the number of IO calls for device reads and writes. An idea I had late in the project cycle was to implement a least-recently used cache of lines of `(nonce, tag)` pairs which are likely to be needed in succession. Obviously the whole header can't be pulled into memory as it may grow to over 112 gigabytes. Specifically, upon a read of block `n`, the nonces and tags for a range of blocks including `n` could be pulled into memory and maintained with LRU replacement. A tunable parameter, for example, could dictate the size of the LRU cache, and performance comparisons against various benchmarks could be used to select the optimal cache line size (i.e. how many `(nonce, tag)` pairs are mapped into memory upon accessing a pair in a given line).

My implementation can currently be integrated with a block device abstraction which can be used on *NIX machines which can offer standard file descriptor abstractions. Future work to integrate this system directly with kernel block device drivers as a configurable option, or with boot-time disk decryption software would take this project from modular but isolated to usable and integrated.

The Qubes OS team have implemented networking driver isolation through use of VT-d, and are currently working on similarly isolating access to the disk bus to de-privilege access to the filesystem. While this work could be effectively utilized in the privilege Dom0 domain, principle of least privilege as well as separation of Dom0 from critical, complex procedures would dictate that the ideal implementation would run this projects code in the untrusted filesystem access domain. Future work could integrate with modifications on the Linux device mapping system in a paravirtualized environment for optimal performance in such an implementation.

Conclusions

The major takeaway from this work for me was that the problems associated with AES-XTS full-disk encryption might be solvable with just a few additional security requirements. I hope to see further work in this direction for the most paranoid users who are perhaps willing to even give up some amount of disk performance in exchange for better security guarantees than AES-XTS can provide especially with the increased popularity of SSDs which may increase the exposure for users depending on deterministic encryption for confidentiality.

References

- *H. Bock, A. Zauner, S. Devlin, J. Somorovskyand, and P. Jovanovic* “Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS” 2016
- *N. Ferguson* “Authentication weaknesses in GCM” 2005
- *P. Rogaway* “Evaluation of Some Blockcipher Modes of Operation” 2011