Initial Analysis of GBDE

Roland C. Dowdeswell
elric@imrryr.org

Abstract

GBDE was proposed at BSDCon 2003 as a disk encryption mechanism for FreeBSD GBDE [2]. GBDE's author claimed that with GBDE it would take 2^{128} steps to compromise one sector, 2^{256} steps to compromise the entire disk and that it would take 2^{384} steps to compromise the disk if the "lock sectors" were destroyed.

We find lower numbers for each of these three as well as fundamental transactionality issues.

1 Introduction

This is only a rough draft. A very rough draft. I will be improving the mathematics in here over time. I make a number of empirical arguments which should be a little stronger, but I wanted to give an idea of where I am going before I spent a lot of time on it. Some of the assertions made are a little strong, and are guesses. I try to call these out by putting them in paragraphs comprised of italics such as this one.

GBDE is a pseudo–disk driver presenting the image of a disk to other operating system facilities such as file systems anddatabases. It is similar in construction and goals to NetBSD's CGD [1], OpenBSD's svnd(4) or Linux's crypto–loop.

At BSDCon 2003 and BSDCan 2004 GBDE's author claimed that a brute force attack of the entire disk would take at least 2^{256} steps and that if the "lock sectors" are destroyed it would take at least 2^{384} steps to crack the disk[3][4].

2 The Cryptography

In this section we analyse some of the cryptographic properties of GBDE.

We begin with a quite terse description of how GBDE works, we expect readers to also read "GBDE—GEOM Based Disk Encryption"[2] and assume knowledge of that paper.

We follow this with a number of sections which analyse the methods.

2.1 Brief overview of GBDE's encryption

In this description, we elide any details which are not relevant to the weaknesses discussed below, such as offsetting the encrypted portion within the partition, etc. We discuss some of these details in Section 2.6.

To turn a pass phrase P into a key:

1. compute $(s_1, s_2, s_3) = SHA2/512(P)$, where s_1 is the first 128 bits, s_2 is the second 256 bits and s_3 is the remaining bits,
2. either the first sector of the disk or a file is decrypted with AES128 s_1 to reveal the location of a "lock sector",
3. the "lock sector" is decrypted with AES256 and s_2, and
4. the contents of the "lock sector" are shuffled with the bits in s_3.

The lock sector contains (amongst other things) the master key M and the salt S.

Next we have key–key sectors which contain encrypted keys for other sectors. Each key–key is encrypted with AES128.

We define $i(k, n)$ to be inserting n into the middle of k (when considered as bit streams). We define $f(k, m)$ to be using each byte of k to pick a byte from m as a direct index, m is assumed to be exactly 256 bytes long.

Now, if we let n be the sector address of the key–key sector we compute its key k_2 as follows:

1. $k_1 = MD5(i(S, n))$,
2. $k_2 = MD5(i(f(k_1, M), n))$,

The resulting k_2 is the key–key (AES128) used to decrypt the key which is in turn a 128 bit AES key which decrypts the content of its data sector using CBC mode.

This is a very terse description of the algorithm, please cross-reference "GBDE—GEOM Based Disk Encryption"[2] for a better description.

We shall continue using the terminology from this section in the rest of the paper, for brevity.

2.2 Dictionary attacks

GBDE provides no protection against dictionary attacks. Dictionary attacks are the quickest and most likely to yield results. With today's compute power it is quite feasible to perform a dictionary attack on GBDE.

GBDE's author claims in Section 9.3 that users should use 2-factor authentication if they need security. Although 2-factor authentication does raise the bar, it should not be viewed as a replacement for providing protection against dictionary attacks. If we consider using GBDE on a laptop, it is likely that the physical token will be quite near the laptop at any given time. The theft of both of them may be more difficult that the theft of only one, but not substantially so.

In short physical tokens are not a substitute for pass phrases, they should be used in conjunction with pass phrases to enhance security.

GBDE's author claims that worst case to try a pass phrase is

$$W_{SHA2/512} + W_{AES128} + \\ W_{disk\_read} + W_{AES256} + W_{MD5}$$

and the best case is:

$$W_{SHA2/512} + W_{AES128}$$

But, there is a problem with the best case analysis which is that the $W_{SHA2/512}$ can be performed off line since the pass phrase is not salted before the hash is used. That leaves us with a best case of only:

$$W_{AES128}$$

Using s_1, we decrypt the first sector of the disk to determine the location of the lock sector. It is encoded as a 64 bit integer which specifies the byte offset into the disk. If we assume that the disk is 1TB, then only 1 in 2^{16} of these will be valid. So, we have the best case almost all the time.

We can easily eliminate the potential disk seeks with memory for time trade offs, so we shall ignore them. And the other work that we need to do in the worst case is dwarfed by how infrequent the worst case actually is.

So, the work required to perform a dictionary attack is only:

$$W_{AES128} + 2^{-16}\text{worst case} \simeq W_{AES128}$$

Now, let's consider what this means on modern hardware:

Pass phrase strength	Time to crack in days
2^{30}	0.12
2^{35}	3.97
2^{40}	127.5
2^{45}	4072

The times were calculated on an IBM ThinkPad T41 1.7GHz Pentium M which is not the fastest computer money can buy. It can perform $100,000$ setkey/encrypt operations per second.

There is also another dictionary attack which can be performed partially off line, using s_2 and s_3 to try to break the "lock sector". If there is an entirely predictable ciphertext block in the lock sector, this could expand into an off line attack.

We note that the contents of the lock sector do contain a lot of reasonably predictable data such as the apparent sector size, first sector, last sector. It also contains quite a bit of random data such as the master key and salt. If we can predict the first ciphertext block worth of this data, then an off line dictionary attack might work. Since the pass phrase includes a "key" which shuffles the data, we expect that this would work for some pass phrases but not others. The condition would be that the pass phrase would need to shuffle the contents of the lock sector leaving predictable data in the first 128 bits. For the passphrases which do work, we can then build a hashed database of the first ciphertext block of the lock sectors. We then simply scan the disk for matches and attempt the corresponding pass phrase.

This might be worth further examination.

2.3 Weak master keys

From Section 2.1, we have f which transforms one set of bytes into another set by indexing into the master key M. M is comprised of pseudo–random bytes, generated when GBDE is first configured.

It is reasonably obvious that even if the bytes in M are uniformly distributed, the bytes in the output of $f_1(x) = f(x, M)$ will most certainly not be uniformly distributed.

Consider M consisting of all 1's. This is a perfectly reasonable, if unlikely random value. In this case, f_1 will always return all 1's.

We generated 10000 random streams of 256 bytes and noted that the number of distinct bytes in each ranged from 142 to 181. With results clustering around the mode 162.

This is an empirical argument that we will replace with a proper one later.

2.4 Recovering keys from key–key sectors

We can use our discoveries about f from Section 2.3 to order our brute force attack in such a way that it is likely to yield results more quickly.

From Section 2.1, we start the attack in the step used to obtain k_2. We do not presume knowledge of k_1 at all, but rather in:

$$k_2 = MD5(i(f(k_1, M), n))$$

We would use the predictable qualities of the output of f to construct an attack.

For the extreme example, consider if all of the bytes in M are exactly the same. In this case, f would always return the same result. k_2 would always be the same. If we knew that M had only one distinct byte beforehand, then we would be able to obtain access to the disk in only 2^8 steps.

Also, consider that if we were to know which m bytes were contained within M then we could easily construct a compromise with a worst case of success m^{16}. If we

M bytes	Probability
256	0.619708
200	0.540031
180	0.503249
160	0.46049
150	0.436454
140	0.410389
128	0.376133
64	0.129012

Figure 1: Distinct bytes in M against the probability that the output of f will contain only distinct bytes

also knew the multiplicities of each of the values in M we could reduce the search further.

Let us consider what we can say if we know nothing about the contents of M except that it is a randomly generated string of bytes where no attempt to suppress duplicates is made.

If M has m distinct bytes there are $P(m, 16)$ possible outputs with distinct bytes out of a total of m^{16} total possible outputs. The probability that the output will contain only distinct bytes is therefore,

$$\frac{P(m,16)}{m^{16}}$$

Please refer to figure 1 for the probabilities given various m.

We can immediately see that, e.g. for $m = 64$ if we choose only values for the output of f that contain repeated bytes then we have a 87% chance of being right in $2^{128} - P(256, 16)$ steps. This is substantially better than the normal case where we have only a 38% chance—and we have just searched only 38% of the keyspace.

We can use this to construct an attack by trying values for $f(k_1, M)$ that contain more repeated bytes than one would expect from the output of a uniform distribution.

This description needs quite a bit of work. I need to figure out when to quit, etc... There is also the open question of what the optimal number of repeated bytes for which to search is, how the search should be structured, etc... We have only demonstrated that we're more than likely going to discover the key a little earlier than we would expect, i.e. we have a 69% chance of discovering the key within the first 38% of the search using this method.

For each guess g at the output of $f(k_1, M)$ we:

1. compute $MD5(i(g, n))$ for the key–key,
2. decrypt the key with the key–key, and
3. verify the key in it by decrypting and verifying the sector.

Each time we succeed we gather statistical information about M which can be used to speed up further calculations.

If we know m the number of distinct bytes in the master key and we know what each of those bytes is, then we can compromise each sector in m^{16} steps. After compromising enough sectors in this way, we will be able to statistically analyse this.

From Section 2.3, with our 10000 randomly generated keys, the weakest one was 142 distinct bytes. This comes to an effort of 142^{16} which is approximately $2^{114.5}$.

If we know the multiplicity of each byte, then we can structure our search to do even better than this because we know the relative probabilities that each byte will appear.

We will come up with the number which is presumably reasonably smaller than $2^{114.5}$ which represents the effort to brute force a sector given knowledge of the bytes and their multiplicities.

After compromising 2^{10} sectors, we should have a reasonably good idea about m, the byte values in M and their multiplicity. Considering $m = 142$, we can then compromise the entire disk in less than:

$$2^{10}2^{128} + n2^{114.5}$$

where n is the number of sectors in the disk.

2.5 From salt to master key

If we assume that we know S, then how could we retrieve M? It turns out that we can tease M out, by using the divide–and–conquer strategy outlined below.

We shall use a 1TB disk as an example. We have 2^{30} sectors and hence we have 2^{30} key–keys.

The algorithm:

1. compute k_1 for each of the key–keys,
2. repeat until M is fully known:
   (a) find the k_1 which has the least unknown distinct byte values,
   (b) crack the key–key discovering the bytes of M indexed by k_1 ,

Cracking the key in step 2b is substantially easier than a brute force of 2^{128}. If we assume that the number of distinct bytes in the best k_1 is n, then we can crack its key–key in 2^{8n} steps.

Once we have the first 16 bytes of M, each addition iteration of the loop takes much less time. So, the total time investment is $O(2^{8n} + noise) = O(2^{8n})$.

Compute the least number of distinct bytes I am likely to find in 2^{30} random 16 byte strings. I think that it will turn out to be around 10 in which case we can go from the salt to the master key in 2^{80} steps.

The same sort of analysis as applied in Section 2.4 can be applied to speed the attack, but it is less effective since there are less bytes chosen from M.

2.6 Other information

EDITOR: need to think of a reasonable title for this section.

There are many pieces of other information which various people have claimed are difficult to find and hence make brute forcing the disk impossible. GBDE places the key–key sector at an offset for the beginning of each group of 32 sectors. The actual disk is offset into the partition and random bits placed around it, etc.

Finding the key–key sector is at most 2^5.

The extent of the encrypted disk is not nearly as random as one would believe. They typical user is not willing to sacrifice much space. It would be interesting and we believe possible to define an algorithm that could perform a modified binary search for the extent of the disk.

GBDE's author mentions some of the information that a cleaning lady can gain, but does not mention that all of this information could be garnered by a cleaning lady. If I see a single disk write it updates two disk sectors. Now I know that the key–key sector is one of them.

We note that disk analysis can provide all of the advantages of a cleaning lady. Hence an attacker with reasonable resources should be presumed to be able to gather all of this information by examining the physical media.

2.7 Decrypting one file

To find one modest sized file on a GBDE partition, one would need to:

1. decrypt and verify the superblock,
2. for i in number of directories deep the file is:
    (a) decrypt the directory,
    (b) find the inode associated with the file or subdirectory in question,
3. decrypt the file's blocks.

If we assume that we need n sectors from the disk to find what we are looking for, then the effort required will be $n2^{128}$ for a strict brute force attack.

It has been asserted in various forums that to perform this attacks, one would need to tackle the problem of an enormous number of false positives and have enormous amounts of data storage.

False positives will be quite unlikely, and in the event that there is one at most it will add a few bits of effort to the search. If blocks are discovered to be false positives, one can just backtrack. The only possible storage requirement would be a small stack.

We can use the knowledge from Section 2.3 to structure the search in a much more efficient manner.

3 Reliability

Because GBDE turns one sector write into two sector writes that is a write to the key–sector and a write to the sector itself, a race condition is introduced where if the OS crashes or if removable media is removed at the wrong time the contents of the sector will be lost.

This can manifest itself during surprising operations, e.g. if a file is read then the atime of the inode is updated. If the OS crashes or if the power fails during this operation then the inode sector will contain random data when the operating system reboots.

File systems are built on the fundamental assumption that sector writes are atomic operations which either succeed or fail but do not fill the sector with random bits.

To rectify this problem, GBDE would need to implement a journal keeping track of outstanding operations and that would reduce its performance substantially.

4 Conclusion

There are number of weaknesses in GBDE which need to be addressed.

Dictionary attacks are the most feasible way to compromise any system which includes a pass phrase. Claiming that the user should use two factor authentication both ignores what users actually do and makes often unwise assumption that the second factor will not be compromised.

The weak master keys need to be eliminated. This could be accomplished by ensuring that all of the bytes in the master key are distinct. We understand that this makes the compromise described in Section 2.5 easier, but that attack is still more difficult than using brute force to compromise each sector.

It would be substantially more prudent to replace f with a function that has some cryptographic properties. It would also be prudent to eliminate the divide-and-conquer compromise by ensuring that the entire master key is used in deriving the keys for the key–key sectors. Both the domain and the co–domain of f are subject to analysis. We do not believe that we have discovered the only problem, in fact quite to the contrary.

The claims about GBDE's resistance to brute force analysis should be adjusted. One cannot use a 128 bit cipher with different keys for each sector and claim much better than 2^{128} resistance to brute force attacks. If GBDE's author wants to claim greater protection then he must use a cipher with more bits on each sector.

References

[1] Roland C. Dowdeswell and John Ioannidis. The cryptographic disk driver. In *USENIX Annual Technical Conference, FREENIX Track*, pages 179–186. USENIX, 2003.

[2] Poul-Henning Kamp. Gbde-geom based disk encryption. In *BSDCon*, pages 57–68. USENIX, 2003.

[3] Poul-Henning Kamp. Making sure data is lost. http://phk.freebsd.dk/pubs/bsdcon-03.slides.gbde.pdf, 2003.

[4] Poul-Henning Kamp. Making sure data is lost. http://www.bsdcan.org/2004/papers/gbde.pdf, 2004.