Lanarea DF

Haneef Mubarak

Abstract

The flaws of current and previous methods of utilization, storage, and conversion of passwords into secure keys, namely plaintext, hashing, and key derivation are explored. A novel, yet simple key derivation algorithm, Lanarea Derivation Function (Lanarea) is explored in depth along with its strengths and weaknesses and the features that it utilizes to solve the flaws exposed by previous algorithms. Individual sections of the algorithm are dissected in an attempt to explain their necessity. Finally, reasons as to why the algorithm cannot be easily and/or efficiently brute forced with commodity hardware (GPGPUs), specialty hardware (FPGAs), and custom hardware (ASICs) are analyzed.

Introduction

Storing passwords in plaintext is highly insecure, because if an attacker breaks into the database or file used to store the passwords, they can use the resulting passwords with any other systems that any given user might have an account on. The simplest method of solving this problem, is to use a cryptographic hash with the password as an input (like SHA1 or BLAKE2); the output is stored, and to verify a user, the password is hashed again at which time the resulting output is checked against the output stored in the database. This is better because now an attacker cannot easily acquire the password for a user from the database.

However, an attacker with sufficient compute power can simply brute force a hash by trying lists of combinations of common words and techniques; essentially constructing passwords similar to the manner that most people create theirs. Since any password translates to a fixed output, once a single password/hash combination is broken, all of the hashes that match that combination are also instantly known [2]. The simplest method of combatting this is to use a salt, but then another issue remains: hashes are designed to be fast [3].

Given that hashes are designed to be fast, even though the number of possibilities

increases vastly, using a GPGPU or even a set of GPGPUs, massive array of passwords can be cracked simultaneously in negligible time [1]. The solution to this is to use, what is known as a Key Derivation Function (KDF).

A KDF is designed to take time to compute; that way, attackers are slowed in brute forcing passwords. Common KDFs include PBKDF2 [4], bcrypt [5], and scrypt[6].

PBKDF2's main advantages are in its simplicity and configurability; but it fails in that it is easily parallelized. Since it is an extremely straightforward, simple algorithm that is being repeated over and over based on the output of the previous algorithm, the task greatly yields itself to GPGPU's and ASICs, especially considering how relatively little memory is required to implement any of the calculations in the algorithm.

bcrypt's main advantages are that it is heavily serial and that it has a long, expensive setup phase. However, it can still be parallelized strongly; instead of parallelizing within the calculation level, GPGPU parallelization can be still be done by calculating several combinations in parallel, since memory accesses are highly predictable. Since bcrypt does not use large amounts of memory either, efficient ASICs can also easily be created for the enormously fast brute forcing of hashes.

scrypt's main advantage is that it requires larger amounts of memory to access. This used to be quite an issue, however, recent developments have resulted in many implementations for GPGPUs [7][8][9] and ASICs are currently available for preorder [10].

Lanarea

Lanarea Derivation Function (pronounced "lon-na-ray-yuh") is a new KDF that is designed to be extremely difficult to execute in parallel on a GPGPU or ASIC. It uses a simple setup algorithm followed by a rather convoluted scheme to rearrange and apply data as the main loop for the calculations, which finally ends with a simple algorithm to extract an output key of any length that is an integer multiple of 32 bytes. The function can take an input password of any nonzero length along with a salt of any nonzero length. It is also configurable, as the pseudorandom function (PRF) used internally can be replaced with any cryptographically secure PRF that takes a variable length input along with a variable length salt and provides an output of 32 bytes. For the default Lanarea function, PRF used internally is BLAKE2b [16].

Description

Setup

Input:

C_r	The memory (RAM) cost of the function
$F\ (input,\ salt)$	A cryptographic pseudorandom function with an optional salt

Constants:

$(p_0 ... p_{127})$	The first 128 bytes of Pi in raw format (0x3243F6...)
$(e_0 ... e_{127})$	The first 128 bytes of e in raw format (0x2B7E15...)

Output:

$(f_{0,0} .. f_{(C_r \times 16),\ 16})$	A matrix for use in the core phase

1. $(i_0 ... i_{127}) \leftarrow (p_0 ... p_{127})$
2. $(i_{128} ... i_{255}) \leftarrow (e_0 ... e_{127})$
3. $(i_{256} ... i_{287}) \leftarrow 0$
4. $m \leftarrow C_r \times 16$
5. $n \leftarrow 16$
6. **for** $x = 0$ to m **do**
7. **for** $y = 0$ to n **do**
8. $(h_0 ... h_{31}) \leftarrow F\ (i_0 ... i_{287})$
9. $f_{x,y} \leftarrow h_y$
10. $(i_{256} ... i_{287}) \leftarrow (h_0 ... h_{31})$
11. **end for**
12. **end for**

This simple setup function does something rather simple: it provides an initialized matrix that is identical for any given size. The purpose of this is that the cost of this could be easily amortized by caching the initialized matrix across invocations of Lanarea. The usefulness of this is that it allows applications such as authentication servers to quickly process passwords while providing negligible benefit to an attacker, as little time ought to be spent in the setup phase.

Core Phase

Input:

C_r	The memory (RAM) cost of the function
C_t	The time cost of the function
$(f_{0,0} .. f_{(C_r \times 16),\ 16})$	The initialized field from the setup function

$(P_0...P_{a-1})$	The password as an octet stream of a length
$(S_0...S_{b-1})$	The salt as an octet stream of b length
F (*input*, *salt*)	A cryptographic pseudorandom function with an optional salt

Output:

$(h_0...h_{31})$	The output of the last hash invocation from within the core function
$(s_0...s_{C_r \times 16 \times 16})$	A stream of octets to be used in the key extraction phase

1. $m \leftarrow C_r \times 16$
2. $n \leftarrow 16$
3. $C_t \leftarrow C_t \times 4$
4. $C_r \leftarrow C_r \times m \times n$
5. $(h_0...h_{31}) \leftarrow F\ (P_0...P_{a-1},\ S_0...S_{b-1})$
6. **for** $x = 0$ to C_t **do**
7. **for** $y = 0$ to m **do**
8. **for** $z = 0$ to n **do**
9. $r \leftarrow (y + h_z)\ mod\ m$
10. $c \leftarrow (r + f_{y,z})\ mod\ m$
11. $r \leftarrow (r + f_{r,z})\ mod\ m$
12. $c \leftarrow f_{c,z}$
13. **if** $(c\ mod\ 2) \equiv 0$ **then**
14. $c \leftarrow ROL\ (c,\ r)$
15. **else**
16. $c \leftarrow ROR\ (c,\ r)$
17. **end if**
18. **if** $(c\ mod\ 4) \equiv 0$ **then**
19. $f_{y,z} \leftarrow (f_{y,z} + h_z)\ mod\ 256$
20. **else if** $(c\ mod\ 4) \equiv 1$ **then**
21. $f_{y,z} \leftarrow f_{y,z} \oplus h_z$
22. **else if** $(c\ mod\ 4) \equiv 2$ **then**
23. $f_{y,z} \leftarrow (f_{y,z} - h_z)\ mod\ 256$
24. **else**
25. $f_{y,z} \leftarrow f_{y,z} \oplus \neg h_z$
26. **end if**
27. **end for**
28. **for** $z = 0$ to C_r **do**
29. $c \leftarrow z\ mod\ n$

30. $r \leftarrow D(z, n)$
31. $s_z \leftarrow f_{r,c}$
32. **end for**
33. $(h_0 \ldots h_{31}) \leftarrow F(s_0 \ldots s_{C_r - 1})$
34. **end for**
35. **end for**

Notes:

1. *Line 14: ROL(value, shift) is a left bitwise rotate (circular shift) on a single byte*
2. *Line 16: ROR(value, shift) is a right bitwise rotate (circular shift) on a single byte*
3. *Lines 21, 25: ⊕is used to signify a bitwise XOR (exclusive disjunction) operation*
4. *Line 25: ¬ is used to signify a bitwise NOT (complement) operation*

5. *Line 30: D(i, w) is an abbreviation of the following expression, used to create an irregular access pattern:*
   $(((w - i \bmod w) \bmod w) + (floor\,(\frac{i}{w}))) \bmod w + (w \times (\frac{i}{w \times w})))$

The core function can be subdivided into a few portions. Lines 8-26 combine the hash with a row within the matrix. Lines 9-12 do pseudorandom read accesses, ensuring that the entire matrix must be stored in random access memory (RAM), thus simultaneously negating any advantages that a given CPU's cache predictor would usually yield. Additionally, this would hinder the use of ASICs to brute force the function, because the pseudorandom accesses cannot be optimized even with prior knowledge of the algorithm, thus forcing continuous memory accesses over a larger area of memory.

Lines 13-17 and lines 18-26, especially the latter, make the use of a GPGPU to accelerate brute forcing extremely difficult, if not worthless, since either branches or a jump table must be used in order to satisfy the use of different functions that are dependent upon a pseudorandom value. Given that GPGPUs are simply massively SIMD[11][12], branching is extremely expensive, and any more than a few occasional branches can negate the accelerations provided by the use of a GPGPU. On a CPU, the branches would also be highly expensive, given that the branch predictor will be unable to predict pseudorandom conditional branches, however, this can be slightly amortized by the use of a jump table [13].

The expression on line 30, as explained in note 5, uses a rather curious access pattern that can be scaled to various square data block sizes [15], which prevents the exploitation of spatial and temporal cache locality [14]. Since the memory cost of the algorithm can be changed, any attacker that wished to utilize an ASIC would be forced to either have small amounts of local memory complemented by additional off-chip memory (with the benefit of more simultaneous cracking cores) at the expense of external memory access cores; or, to have larger amounts of

local memory (with the detriment of fewer simultaneous cracking cores) at the expense of having unused memory until the memory cost is increase and the future loss of the advantages yielded by said ASIC (when the memory cost exceeds the size of the memory available per core on chip).

Key Extraction Phase

Input:

$(h_0...h_{31})$	The output of the last hash invocation from within the core function
$(s_0...s_{a-1})$	A stream of octets of length a
L	The length of the output key in bytes; must be a multiple of 32
$F\ (input,\ salt)$	A cryptographic pseudorandom function with an optional salt

Output:

$(k_0...k_{L-1})$	A key composed of a stream of octets: the result of the *key derivation function*

1. $L \leftarrow \frac{L}{32}$
2. **for** $x = 0$ to L **do**
3. $\quad (h_0...h_{31}) \leftarrow F\ (s_0...s_{a-1},\ h_0...h_{31})$
4. $\quad (k_{x\times 32}...k_{(x\times 32)+31}) \leftarrow (h_0...h_{31})$
5. **end for**

The key extraction function is nothing particularly fancy; it uses the keyed PRF to create a result 32 octet stream, which is added to the output stream of octets until the length requested has been satisfied. Each round uses the result of the previous round as a key.

Analysis

Hand optimized implementations will likely not be more than 10% faster than the compiler optimized+profiled build of the algorithm, because the algorithm is highly serial and there are few possible optimizations.

ASICs/FPGAs

seem to be a pressing concern for security today, as custom circuitry can be optimized to take every advantage of any parallelizations along with any other

available optimizations, after which the circuit can be massively duplicated across the die, yielding absurdly large cracking power to potential attackers. Particular techniques that Lanarea uses to combat this are discussed in the paragraphs following the core function; here is a short summary:

- Pseudorandom Read Accesses (PRA): by doing partially pseudorandom reads across the entire matrix, any ASIC design will be force to either hold the entire matrix within the on-chip ram (thus limiting parallel core count) or to do frequent accesses to external memory banks (thus limiting serial execution speed)
- Irregular Access Pattern (IAP): the irregular access pattern not only forces memory issues similar to the above, it also imposes the requirement for additional circuitry, thus further limiting the number of parallel cores that may be used on a single chip
- Heavily Serial Operations (HSO): the operations used in Lanarea are heavily serial (including the mixing stage - the next operation's random access may be the current operation's write - RAW dependency [17]), and as such, parallelization cannot be done easily within the algorithm, meaning that portions of a circuit will have to be inactive for extended durations and the individual throughput of any given cracking core will be lower in comparison to other algorithms.
- Nonuniform Section Timings (NST): various sections of the algorithm take widely differing amounts of time, thus, pipelining cannot be done using the internals of the algorithm, as there will be major stalls quite frequently. A nonuniform pipeline would not work either, since the mixing stage has pseudorandom time, within a set of limits, and as such, there would still be stalls in a nonuniform pipeline.

These features of the algorithm make the brute force cracking of Lanarea via the use of ASICs rather infeasible.

GPGPUs

are often used to crack KDFs (by trying many passwords in parallel), but attempting to do so with Lanarea would be likely impossible:

- PRA: GPGPUs are designed to process data with uniform memory accesses and to process data within blocks, nonuniform memory accesses are far slower on a GPGPU due to this.
- NST: the SIMD nature of a GPGPU is incompatible with nonuniformity in timing across lanes of execution; stalls and delays would occur quite frequently due to this.

- Pseudorandom Mixing Functions (PMF): GPGPUs' SIMD nature means that they cannot branch like this within *every single byte's mixing round*, and as such, this strongly eliminates the possibility of an efficient GPGPU implementation of a Lanarea brute force cracker.

CPUs

are what Lanarea is designed to run on; nevertheless, Lanarea also tries to ensure that various CPUs of various generations still take roughly the same amount of time per invocation:

- PRA: doing pseudorandom accesses mean that the L1 cache predictor will not be able to correctly predict the data that will be loaded next. This negates the effects of a cache predictor, putting CISC CPUs such as x86/AMD64 on par with RISC CPUs such as ARM in terms of memory accesses (although platforms such as x86 will still have a fair lead due to their large caches).
- IAP: has the same effect on most processors as PRA, for mostly the same reason, thus further pushing complex platforms like x86 (mainly used in client and server computers) onto par with less powerful platforms, like ARM (which is mainly used in mobile and other low-power devices).
- HSO: negates the advantages yielded by the out-of-order, superscalar nature of platforms such as x86 [18], since many instructions will have to wait for previous instructions to complete, thus causing a pipeline stall. Since simpler processors often have shorter pipelines, they gain an advantage here, albeit a small one, which, nevertheless, brings them even closer to par with more complex architectures. Additionally, the serial nature of the algorithm means that the only real method to accelerate execution of the algorithm would be to increase the core clock frequency; this is mitigated by the fact that most modern processors' native clock frequencies all lie within an order of magnitude or so of one another.
- PMF: since the branch predictor cannot predict pseudorandom conditional branches, it is more performant to utilize a jump table. As almost every modern processor has some sort of support for jump tables, this further negates the advantage that complex processors have over simple ones, since their powerful branch predictors become worthless and can even be detrimental in situations like this.

Security

Lanarea itself is secured rather well in that most of its security is dependent upon the PRF used for its internal computations, BLAKE2b. BLAKE2 has been verified quite

thoroughly and appears to be extremely safe to use. In the event that severe issues are found with BLAKE2, Lanarea can be trivially reconfigured to use any other PRF that can accept a variable length input along with a variable length salt and output a 32 octet result.

Lanarea's key derivation is believed to be secure for the following applications:

- Authentication
    - server-client (web, etc.)
    - system level (*nix shadow, etc.)
- Encryption
    - stream (sockets, etc.)
    - long term (disk, flash, etc.)
-

Conclusion

While algorithms such as HMACs, PBKDF2, bcrypt, and scrypt have served us for a long time and served us rather well, it is time for them to retire because there are too many issues that a potential attacker can exploit to allow them to brute force the passwords. A new key derivation function is sorely needed to mitigate this issue, and Lanarea does it rather nicely while also taking care of some of the other issues that the previous KDFs had, such as:

- drastically faster completion on one platform versus another
- easy parallelization that has been exploited by attackers
- complexity that hinders the ease of understanding the algorithm from scratch along with making it slightly more difficult to implement
- mind numbingly boring names
- etc.

Lanarea seeks to solve this issue by providing a platform independent, non-parallelizable, simple, creatively named KDF that executes with relatively consistent speed. Interestingly enough, in the process of designing the algorithm ,it was found that the algorithm *took too long* and was thus modified so it could run in real time.

Miscellanea

At the time of writing, there are no backdoors or weaknesses whatsoever in the algorithm Lanarea that I possess knowledge of that have not been described to the

best of my abilities above.

I, Haneef Mubarak, release any intellectual property that I may hold on the algorithm Lanarea or it implementation into the public domain, where it shall forever remain, available globally on a royalty free basis, to the furthest extent permitted by law. I am fully unaware of any patent or application that covers any portion of the algorithm Lanarea or its implementation.

Acknowledgements

- freenode
    - yarrkov - encouraging me to join the competition and pointing me to resources
    - LadyRainicorn - helping me come up with the name
    - K_F - helping me come up with the name
    - enlga - helping me debug segfaults
    - twkm - reminding me that magic constants are evil
- StackOverflow
    - Adam Burry - helping out with the pattern production code
    - ThoAppelsin - explaining out how the pattern production is done
- The authors of BLAKE2 - putting BLAKE2 in the public domain

Thank you all for all of your help; I really do appreciate it and it means quite a lot to me.

Citations

1. http://blog.codinghorror.com/speed-hashing/
2. http://security.stackexchange.com/questions/33505/why-using-random-salts/33521
3. https://blake2.net/#qa
4. http://crypto.stackexchange.com/questions/3484/pbkdf2-and-salt
5. http://yorickpeterse.com/articles/use-bcrypt-fool/
6. https://www.tarsnap.com/scrypt/scrypt.pdf
7. https://github.com/veox/sgminer
8. https://github.com/Project-Khepri/Scrypt-Kernel
9. https://github.com/darksea/cgminer
10. https://www.kncminer.com/products/titan
11. http://stackoverflow.com/questions/5919172/gpgpu-vs-multicore
12. http://www.ai.mit.edu/projects/aries/papers/writeups/darkman-writeup.pdf
13. http://staff.ustc.edu.cn/~han/CS152CD/Content/COD3e/InMoreDepth/IMD2-Jump-Tables.pdf
14. http://www.cs.uiuc.edu/~snir/PDF/Temporal%20and%20Spatial%20Locality.pdf
15. http://stackoverflow.com/questions/22647907/multidimensional-array-patterned-access/
16. https://blake2.net/blake2_20130129.pdf
17. http://coitweb.uncc.edu/~abw/ITCS3182F09/slides10Z.pdf
18. http://www.gamedev.net/page/resources/_/technical/general-programming/a-journey-through-the-cpu-pipeline-r3115