New developments in password hashing: ROM-port-hard functions

(building upon the ideas of scrypt and security through obesity)

Solar Designer <solar@openwall.com>

@solardiz @Openwall

http://www.openwall.com

November 2012

Historical background

HTTP://OPENWALL.COM/PASS

Concepts to be familiar with:

- Password hashing
- Key derivation function
- Salting
- Password stretching
  - bcrypt, PBKDF2
- Memory-hard functions
  - scrypt

Problem definition

- Password hashing on servers dedicated to authentication (or maybe even to password hashing alone), at an organization large enough to go for this approach anyway
  - Some of the same approaches are also usable/tunable for use on non-dedicated servers - e.g., by operating systems for local users' passwords - but this may be a next step
- Need decent throughput (~1000/s per server) and sane latency (< 100 ms)
- Want to make best use of the server hardware to slow down offline password cracking (and thus reduce the percentage of passwords getting cracked in real-world attacks) should the password hash database (and a local parameter, if any) leak or be stolen

Issues with scrypt for mass user authentication

At low durations and decent throughput, we face two problems:

- Low memory usage: acceptable at 100 ms (~32 MB), way too low at 1 ms
- Limited scalability on multi-CPU/multi-core when we maximize RAM usage
  - Optimized scrypt's SMix achieves a throughput of ~1500/s on a dual Xeon E5649 machine (12 cores, 24 logical CPUs) when running 24 threads (thus, latency 16 ms) at 4 MB/each
  - A cut-down hack (Salsa20 round count reduced from 8 to 2, SMix second loop iteration count reduced from N to N/4) achieves the same at 8 MB

    This is sane speed and sane memory usage, but we want to do better - and we can
  - scrypt paper recommends at least 16 MB
  - scrypt at 128 KB (Litecoin) is ~10x faster to attack on GPU than on CPU

Revising scrypt's ROMix algorithm

Algorithm $\mathbf{ROMix}_H(B, N)$

Parameters:

H	A hash function.
k	Length of output produced by H, in bits.
Integerify	A bijective function from $\{0,1\}^k$ to $\{0, \dots 2^k - 1\}$.

Input:

B	Input of length k bits.
N	Integer work metric, $< 2^{k/8}$

Output:

B'	Output of length k bits.

Steps:

```
1: $X \leftarrow B$
2: for $i = 0$ to $N - 1$ do
3:     $V_i \leftarrow X$
4:     $X \leftarrow H(X)$
5: end for
6: for $i = 0$ to $N - 1$ do
7:     $j \leftarrow \text{Integerify}(X) \bmod N$
8:     $X \leftarrow H(X \oplus V_j)$
9: end for
10: $B' \leftarrow X$
```

◀ Can do it once

◀ Any iteration count

What if we reuse the same ROM across hash computations?

- Not a sequential memory-hard function anymore, but the ROM can be arbitrarily large for any throughput and latency
- Attacker's cost per candidate password tested is in ROM access ports and bandwidth

ROMix algorithm description by Colin Percival, "Stronger key derivation via sequential memory-hard functions", 2009

ROM-port-hard functions

- Not a precise definition, more like word play on Colin Percival's sequential memory-hard functions concept
- We might access only a tiny fraction of array elements per hash computed (vs. scrypt's 100% write, 63% read), but that's OK as long as each element is equally likely to be needed and the access pattern is not predictable
- Because of the above and since the ROM must stay read-only, can't defeat the time-memory trade-off by modifying the array (we could in scrypt)
- However, can defeat the TMTO by pre-filling the ROM differently (not allowing for one array element to be quickly computed from another)

Pros of ROM-port-hardness (vs. scrypt)

- Can use ~1000 times more memory on current server hardware (and require as much memory in each node for efficient attack: anti-botnet)
  - e.g., 10 to 240 GB of ROM (actually in RAM) vs. scrypt's 4 MB or 8 MB per thread
- Can use a server's full RAM capacity even when the current request rate is low (that is, when few hashes are being computed concurrently)
- Excellent scalability to more CPUs and CPU cores
  - We only need to increase the amount of processing between memory accesses to be such that we stay just below saturating the memory bandwidth when all CPU cores are in use (unlike with scrypt, this does not result in reduction of memory usage)
- Can use types of memory other than RAM (e.g., SSDs)

Cons of ROM-port-hardness (vs. scrypt)

- Good scalability of attacks to more computing power (CPUs, CPU cores, GPUs, etc.) while not having to provide more memory
  - However, if the defender stayed just below saturating the memory bandwidth then the attacker may have to provide more bandwidth first
- A custom or otherwise more suitable hardware setup would have a larger number of ports to the same ROM capacity
  - Moreover, those don't have to be ports to the same large ROM - instead, separate smaller ROM banks may be used as long as bank conflicts are fairly rare
  - Yet at below ~1000 cores sharing a ROM the attack speed per die area will be lower than for scrypt, whereas with more cores the attacker will have to provide more interconnect and ROM ports, which will have a cost of its own

The best of both worlds

- We can have a function that is
  - sequential memory-hard with a small amount of RAM (a few MB)
  - ROM-port-hard with a large amount of ROM (many GB)
- Trivial: compute e.g. scrypt and our ROM-port-hard function sequentially (feeding the output of one into the other) or independently (then combine the outputs e.g. using a fast hash)
  - Drawback: an attacker may use smaller memory machines to compute the scrypt portion
- Smarter: merge the two functions, interleave the memory access types
  - In scrypt terms, this can be done in SMix or BlockMix
  - The block size and relative frequency of small RAM and large ROM accesses may be tunable in case different memory types are being used or caching plays a role

Blind hashing

As proposed by Jeremy Spilman after the LinkedIn password hashes leak:

- Keep salts in the user records, but disassociate hashes from users
  - To set a password: generate a random salt, compute the hash, store the salt in the user record, but store the hash separately
  - To validate a password: search the entire hashes table for H(salt, password)
- To ensure there's only one active password (prevent backdoor passwords):
  - To set a password: generate two random salts, compute two hashes, store salt1 and hash2 in the user record, store salt2 and hash1 separately
  - To validate a password: look up salt2 by searching the entire hashes table for H(salt1, password), then check if H(salt2, password) matches hash2

    It may be better to define hash2 = H(salt2, hash1, password)

Security through obesity

- Jeremy also proposed that fake hashes be added to make the hashes table arbitrarily large
  - Harder to steal, distribute to attack nodes, search on smaller machines
  - Targeted offline attacks (e.g., on just one target user account) either require all records from the hashes table or will fail for some passwords
- reddit user alex_w dubbed this "security through obesity", a term accepted by Jeremy and later used more broadly by Ethan Heilman
- Jeremy refers to his specific approach as "blind hashing", which is an instance of the "security through obesity" approach

Issues with blind hashing

- It is easy to inadvertently leave clues that would enable an attacker to distinguish real vs. fake hashes or even map users to hashes
  - Filesystems, databases, web servers, etc. might store meta-information such as timestamps, or/and they might reveal likely relevance between records through use of adjacent filesystem blocks, journal records, or/and log file records
  - Fake salts and hashes themselves might be inadvertently distinctive
- Partial hashes, Bloom filter, or even a bitmap are sufficient to rule out most {user, candidate password} combinations - can save space on attack nodes
  - e.g., a 1 GB Bloom filter will rule out 98% of possibilities against a 1 billion entry table, whereas a 250 MB bitmap will rule out 60%
  - on the server, the same 1 billion entry table would be many gigabytes

More issues with blind hashing

- Since we compute two hashes per authentication attempt, the running time and memory cost of each is reduced (a ~4x area-time product reduction)
  - Yet some attacks may work on one of the hashes individually
  - ... and even when not, the lower memory cost helps the attacker

    This can be repaired by revising scrypt to initialize V only when computing hash1, then reuse same V for hash2
- Upgrading hashes to a stronger type is more costly or/and revealing
- Since the attacker will be able to use partial hashes and might not need salt2 anyway, the defender will want to make these as small as practical
  - If taken "too far", this brings up its own difficulties, such as the need to handle occasional partial hash collisions, resulting in greater timing leaks (from extra hash2 computations when hash1 lookup returns multiple salt2 values) and greater susceptibility to DoS attacks

How about obese scrypt instead?

- Computing our ROM content from a site-specific(?) seed value at service startup may take ~1 minute (e.g., 60 GB at 1 GB/s)
- Loading it from an SSD may take ~5 minutes (e.g., 60 GB at 200 MB/s)
  - Alternatively, we can mmap() it and start serving requests right away, getting to full speed in a few minutes as the content gets cached in RAM
- Then we don't have to store the seed value on the server - in fact, we don't have to store it at all (nor to ever have had it)
- The entire site-specific ROM would have to be stolen and distributed to all attack nodes (in addition to them needing this much RAM for sane speed)

Taking obese scrypt a step further

- Besides using an SSD to load our ROM content into RAM, we can keep a larger ROM on SSD - and use it from there
  - This may be achieved with mmap() and making less frequent, larger block size accesses
  - May want to prefetch data on a previous loop iteration, to avoid stalling computation
    - Consider: madvise(), aio_read(), helper thread
- If there's no seed value stored on the server, the intruder will have to copy and likely distribute the SSD ROM content to attack nodes
- Multiple SSDs and potentially even a NAS/SAN based on SSDs may be used to make the ROM even larger
  - Not practical? Compared to blind hashing with database size similar to our SSD ROM's, this may be more practical and it has more obvious properties

Issues with ROM on SSD

- SSD read disturb errors are a potential concern
  - The firmware would presumably prevent these by rewriting or relocating the block after "too many" reads have been made
  - Theoretically, these rewrites could in turn wear the SSD out
  - A back of the envelope calculation suggests that theoretically it'd take on the order of a million years until this problem would occur, assuming smart firmware
- If the firmware maintains per-block read counts and they're retrieved by an intruder, this may potentially allow for early-reject of some candidate passwords (block never accessed? this password must be wrong!)
  - Mitigation: start using the ROM SSD half way through computation of a hash

Light use of ROM on SSD

- Rather than use our SSD ROM throughout hash computation, we can access it just once before a final cryptographically secure step (e.g., before the final PBKDF2-HMAC-SHA-256 invocation in a revision of scrypt)
  - This is much simpler to implement and it avoids the issues/concerns with using SSDs
  - It is friendly towards other uses of the same SSDs since we would only be making ~1000 requests/s from each machine (one request per hash computed), which is more than an order of magnitude below SSDs' IOPS capacity
- The attacker will need to have access to a copy of the SSD ROM for offline password cracking, but will not need to distribute it to attack nodes

Questions?

Solar Designer <solar@openwall.com>

@solardiz @Openwall

http://www.openwall.com