Password cracking: past, present, future

A talk by

Solar Designer <solar@openwall.com>
@solardiz

https://www.openwall.com
@Openwall

Credits: CIQ/Rocky Linux

May 10-11, 2024
Berlin, Germany

INTRODUCTION

Why passwords?

* From a defensive perspective, "password" should usually mean "passphrase"
  + We proceed with "password" as a general term here
* Passwords remain a distinct and ubiquitous authentication factor
  + "Something you know" in 2FA/MFA
* Passwords are widely used to derive encryption keys for data or other keys
* Passphrases may also be used to derive private keys e.g. for wallets
  + Generated seed phrases are commonly used, but extra words may be added
* Similar concerns and techniques apply to other hashed low entropy data

Why crack passwords?

* Authentication - prevent or gain access to systems

  + Security audits, penetration tests
  + Access recovery (alternative to password reset)
  + (Un)authorized

* Key derivation - recover or gain access to data, other keys, or funds

  + By owner, heir, service provider, law enforcement, anyone (un)authorized

* General

  + Enhancement of breached password lists, training and test sets for tools
  + Research projects, contests, hobby, historical preservation
  + Analysis of malware and backdoors, OSINT, recovery of other hashed data

Scenarios

* Focus of this talk
  + Offline against a local copy of hashes, traffic dump, encrypted data
  + Optimization

* Also closely related
  + Online against a remote system - e.g. Hydra by van Hauser / THC (2001+)

* Not so closely related
  + Online against a remote system with unprivileged local access
  + Side-channel inference (network or local keystroke timings, etc.)
  + Physical against a (tamper-resistant) device

* Mostly unrelated, although these may be best when they work
  + Password reset, plaintext password leak/extraction, password check bypass
  + Crack derived key directly

Targets

* Focus of this talk is tools usable for authentication passwords

* There are also tools specialized for certain other targets such as encrypted files, filesystems, etc., but these are mostly beyond scope

* Current most flexible password crackers - John the Ripper and hashcat - are not limited to authentication passwords, so most of the information here also pertains to password cracking/recovery for other supported targets

* John the Ripper started to gain support for what we call "non-hashes" with Dhiru Kholia's Google Summer of Code project under Openwall (2011)
  + We process non-hash files via *2john tools, which extract pseudo-hashes
  + Some reveal the encrypted data, better ones don't; btcrecover's also vary

* hashcat followed suit and accepts pseudo-hashes output by *2john

Optimization

* Speed (parallel throughput vs. hardware, maintenance, energy cost)
  + of candidate password generation (vs. focus)
  + of duplicate candidate password avoidance or suppression (vs. uniqueness)
  + of hashing or key derivation (vs. memory, storage, and reliability)
  + of matching against loaded hashes or checking for correct decryption
  + instantaneous (async) vs. average until completion (have to sync)
* Memory and storage requirements (vs. speed, focus, uniqueness)

* Focus (optimal search order)
  + vs. candidate password generation speed and parallel processing
  + vs. ease of reasoning (for checkpointing, reporting, exclusion)
* Targeting (custom candidate password streams per target)
* Uniqueness (no or few duplicate candidate passwords for same target)

* Feedback loops and workflow (refocus based on passwords cracked so far)

PASSWORD AUTHENTICATION TIMELINE

Plaintext password storage (1960s to early 1970s CTSS, TENEX, Unix)

```
password
   |
   v
password store

password   password store
     \ /
      v
   compare
is it timing-safe?
```

Plaintext password storage (1960s to early 1970s CTSS, TENEX, Unix)

* On CTSS, "one afternoon [...] any user who logged in found that instead of the usual message-of-the-day typing out on his terminal, he had the entire file of user passwords"

Fernando J. Corbato, "On Building Systems That Will Fail", 1991

The problem was a text editor temporary file collision, "early 60's" to "1965" by different sources

* TENEX had a character-by-character timing leak exacerbated by paging

* "The UNIX system was first implemented with a password file that contained the actual passwords of all the users"

Robert Morris and Ken Thompson, "Password Security: A Case History", 1978

Password hashing (early 1970s Multics)

* "Multics User Control subsystem stored passwords one-way encrypted [...] I knew people could take square roots, so I squared each password and ANDed with a mask to discard some bits."

* After successful break by the Air Force tiger team doing a security evaluation of Multics in 1972-1974, "we quickly changed the encryption to a new stronger method"

Tom Van Vleck, "How the Air Force cracked Multics Security", 1993

Password hashing (mid 1970s Unix)

* crypt(3) of Unix up to 6th Edition inclusive reused code from an "encryption program [that] simulated the M-209 cipher machine used by the U.S. Army during World War II. [...] the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key."

* "The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed."

* "It takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations."

Robert Morris and Ken Thompson, "Password Security: A Case History", 1978

Password hashing (mid 1970s Multics & Unix)
2000s web apps & Windows

```
password
        \
         v
   hash function
         v
password hash store
```

```
password   password hash store
       \            /
        v          /
  hash function   /
                \ /
                 v
               compare
```

Password hashing (late 1970s Unix)

```
password   new salt
policy check?\ /      |
             v        |
slow hash function    |
             v        v
password hash store
```

```
password   password hash store
       \ /(salt)    /(hash)
        v          /
slow hash function   /
                \ /
                 v
               compare
```

Password hashing (1990s BSDI, bcrypt, PBKDF2)

```
password   new salt & cost
        \ / (aka setting)
         v       |
tunably slow hash function |
         v       v
password hash store

password   password hash store
        \ /(setting) /(hash)
         v          /
tunably slow hash function   /
                    \ /
                     v
                  compare
```

Password cracking: past, present, future

Password hashing (2010s+ scrypt, Argon2, ...)

```
password   new salt, costs
        \ / (aka setting)
         v       |
memory-hard hash function |
         v       v
password hash store
```

```
password   password hash store
       \ /(setting) /(hash)
        v          /
memory-hard hash function   /
                  \ /
                   v
                compare
```

Other authentication methods (1990s+)

The examples so far involve the plaintext password being sent to the server (hopefully over a transport security layer), but authentication methods not requiring that exist. Usually they are also susceptible to offline password cracking against certain authentication material:

* Challenge/response (many kinds): network sniffed challenge/response pairs
  + Even worse, POP3 APOP and CRAM-MD5 require plaintext-equivalent storage

* Kerberos: TGTs, AFS user database

* S/Key, OPIE: skeykeys file

* SSH: passphrase-encrypted private key, hashed known_hosts

* SRP (and other PAKEs): verifiers

PASSWORD CRACKING SPEED

Newsgroups: net.general
Date: Thu Jan 6 08:02:37 1983

We proudly announce
The Second Official
UNIX PASSWORD CRACKER CONTEST

Submit your ingenious /etc/passwd password cracker program (source code) to the undersigned by January 31, 1983. We will test all programs for speed, portability, and elegance, on verious Unix versions and on different machines. A manual page and a short writeup explaining the algorithm is a plus. The writers of the best three programs will win the Grand Prize, The Super Grand Prize, and The Ultra Grand Prize (and world-wide, ever lasting fame).

Ran Ginosar, Computer Technology Research Center,
Bell Labs, Murray Hill.

Password cracking (1980s, unoptimized)

```
* For each user's hash
  + For each candidate password

      candidate password   password hash
                        \ /(salt)    /
                         v          /
                   hash function   /
                                 \ /
                                  v
                               compare
```

Password cracking (unsalted, semi-optimized)

* For each candidate password

```
candidate password   password hash(es)
                  \            /
                   v          /
             hash function   /
                            \ /
                             v
            one-to-many compare
```

* We've amortized the cost of hashing, reusing the result of each computation

Password cracking: past, present, future

Password cracking (early 1990s, salted, semi-optimized)

* For each candidate password (groups of more likely passwords first)
  + For each salt

```
candidate password   password hash(es) that use the current salt
                   \ /(salt)    /(hashes)
                    v          /
              hash function   /
              optimized    \ /
                            v
             one-to-many compare (initially just a loop)
        becomes one-to-one when each salt is unique, as they should be
```

* We can no longer amortize the cost of hashing when salts are globally unique

Password cracking: past, present, future

Password cracking (late 1990s, salted, mostly optimized)

```
* For each candidate password group (those of more likely passwords first)
  + For each salt

    candidate password(s)   password hash(es) that use the current salt
                         \ /(salt)    /(hashes maybe partially reversed)
                          v          /
       many-to-many hash function   /
       cost-amortizing, optimized\ /
       parallelized               v
                 many-to-many compare
          becomes many-to-one when each salt is unique, as they should be

* Amortizing means some parts are computed less often and their results reused
  + "Key setup" and/or "salt setup" may be moved to an outer loop
  + "Finalization" may be reversed at startup and then not performed at all
```

Password cracking many-to-many hash comparison

* If few hashes loaded for the current salt, compare or eliminate directly

* If many hashes loaded for the current salt (1997+ hash tables in JtR 1.2+)

```
 partial hashes for comparison against those being cracked (for current salt)
       v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v
   +-----------------------------------+-------------------------------------+
   1. Prefetch needed elements of ---> |                per-salt             |
   2. Reject hashes not in ----------> | sparse bitmap(s) or Bloom filter    |
   3. Complete computation of the rest | or/and (perfect) hash table, etc.   |
   +-----------------------------------+-------------------------------------+
                                 remaining full hashes
                                        v v
   4. Look up in the final data structure(s), which can start similar to the
      above or be as simple as a linked list right away
```

Password cracking cost reduction

* When we amortize cost, we reduce total cost to achieve the same results
  + We do it e.g. through reducing the total amount of computation

* For well-suited hashing schemes, very little computation can be amortized
  + However, that's not the only way to reduce cost

* Besides computational complexity, the other major metric is space complexity

* Real-world costs may be incurred for hardware, maintenance, energy, etc.
  + These costs are related to both computational and space complexities, as well as to real-world constraints, which may vary by attacker
  + For example, "how many CPUs and how much RAM is occupied for how long, and do we readily have those or do we have to acquire them?"

... and then it might not be CPUs anymore

Password cracking on GPUs (2007+)

* Pioneered by Andrey Belenko of Elcomsoft
  + Initially for NTLM, LM, and raw MD5 hashes, achieving 100M+/s
    + Beyond reach of existing CPU software (except for Cell such as in PS3)

* Andrey and others improved the speeds and implemented further (non-)hashes
  + Whitepixel by Marc Bevand: 33.1 billion/s against one MD5 hash on a sub-$3000 4x dual-GPU HD 5970 computer (2010)
  + oclHashcat-lite by atom: 10.9 billion on one dual-GPU HD 6990 (2012)
  + oclHashcat-plus made GPUs usable almost as completely as CPUs (2012)
    + Closed source at first, tweeted MD5 of "hashcat open source" in 2015

* John the Ripper patches for some hashes in 2010 and 2011, integrated into 1.7.9-jumbo-6, was first to implement bcrypt, sha512crypt (2012)

* hashcat 6.2.6: 164.1 billion/s against one MD5 hash on one RTX 4090 (2022)

Password cracking cost reduction through parallel processing

* Parallel processing during defensive hashing or key derivation is limited

* Parallel processing potential during password cracking is "unlimited"
  + Yet efficient designs that also satisfy other goals are far from trivial

* Thus, attack duration can be "arbitrarily" reduced through addition of parallel processing elements (CPUs/SIMD, more/larger GPUs/FPGAs/ASICs)
  + along with accordingly more memory
    ... unless the hashing scheme allows for memory cost amortization
    + Most older schemes don't use much memory anyway
    + Most modern schemes should avoid this, which they do to varying extent

* Parallel processing doesn't reduce the amount of computation, but
  + it reduces the amount of time for which other resources are held and/or
  + it amortizes their cost (e.g. a CPU alone vs. the CPU+GPUs per chassis)

Password cracking cost reduction through time-memory trade-off (TMTO)

* With no salts or few commonly used values, it may help to precompute and partially store the hashes to bypass most computation in future attacks
  + QCrack (1995-1997) for DES-based crypt(3) could save ~1 day per disk
  + Rainbow tables (Philippe Oechslin, 2003 building on Martin Hellman, 1980)

* It may be possible to compute a function in less time by using more memory
  + Matt Bishop, "A Fast Version of the DES and a Password Encryption Algorithm", 1987 uses larger/combined lookup tables (up to 200 KB total)

* Conversely, it may also be possible to compute a function in less memory by throwing away and recomputing intermediate results when needed
  + scrypt is deliberately friendly to this trade-off, which crackers use

* With many same-salt hashes, early-rejection rate may be increased (and thus further computation and matching reduced) with larger and sparser bitmaps

Password cracking cost metrics

* For a given performance ({password, hash} tests per time, maybe amortized)
  + Hardware: ASIC die area, mm^2
    + for a certain design at a certain clock rate in a certain ASIC process
  + Power, W
    + not a cost per se, but for lengthy attacks translates into energy cost
    + correlates with die area

* For a given attack ("test these candidate passwords against these hashes")
  + Hardware: ASIC die area and time product (area-time, AT), mm^2 * s
  + Energy: power and time product, J = W * s
    + correlates with AT, letting estimate relative costs in AT terms alone

* This is state of the art theoretical model for informing defensive designs
* Hardware and energy may have monetary costs, but not always to the attacker
* Real-world attackers' costs may vary greatly e.g. due to existing hardware

Parallelized hash function (originally memoryless)

```
                    candidate passwords
     v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |  core  |  core  |  core  |  core  |
+--------+--------+--------+--------+--------+--------+--------+--------+
     v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v v
  hashes for comparison against those being cracked (for current salt)
```

32 hashes in parallel in the same amount of time that a defender needs for one

Parallelized hash function (amortizable memory-hard)

```
                          candidate passwords
                     v v v v v v v v v v v v v v v v
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |                                    |
+--------+--------+--------+--------+                                    +
|  core  |  core  |  core  |  core  |                                    |
+--------+--------+--------+--------+                 memory             +
|  core  |  core  |  core  |  core  |                                    |
+--------+--------+--------+--------+                                    +
|  core  |  core  |  core  |  core  |                                    |
+--------+--------+--------+--------+--------+--------+--------+--------+
                     v v v v v v v v v v v v v v v v
  hashes for comparison against those being cracked (for current salt)
```

16 hashes in parallel in the same amount of time that a defender needs for one

Parallelized hash function (parallelizable memory-hard)

```
                         candidate password
                                  v
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |  core  |  core  |  core  |                                   |
+--------+--------+--------+--------+                                   +
|  core  |  core  |  core  |  core  |                                   |
+--------+--------+--------+--------+                                   +
|  core  |  core  |  core  |  core  |                                   |
+--------+--------+--------+--------+                                   +
|  core  |  core  |  core  |  core  |                                   |
+--------+--------+--------+--------+--------+--------+--------+--------+
                                  v
    hash for comparison against those being cracked (for current salt)
```

1 hash in 1/16 of the amount of time that a defender using one core would need

Parallelized hash function (sequential memory-hard, e.g. scrypt)

```
                          candidate passwords
                                  v v
+--------+--------+--------+--------+--------+--------+--------+--------+
|  core  |                         |  core  |                         |
+--------+                         +--------+                         +
|                                  |                                  |
+               memory             +               memory             +
|                                  |                                  |
+                                  +                                  +
|                                  |                                  |
+--------+--------+--------+--------+--------+--------+--------+--------+
                                  v v
  hashes for comparison against those being cracked (for current salt)
```

2 hashes in parallel in the same amount of time, but we need 2x more memory

Parallelized hash function (sequential memory-hard + ROM-port-hard)

```
                          candidate passwords
                                  v v
+--------+--------+--------+--------+--------+--------+--------+--------+
|        |  core  |                                                     |
+  RAM   +H------H+                                                     +
|    port=#      #=port                                                 |
+--------+        |                yescrypt ROM 100+ GB                 +
|    port=#      #=port                                                 |
+  RAM   +H------H+                                                     +
|        |  core  |                                                     |
+--------+--------+--------+--------+--------+--------+--------+--------+
                                  v v
  hashes for comparison against those being cracked (for current salt)
```

2 hashes in parallel in the same amount of time, but we need lots of memory

Segmentation fault (core dumped)

In these illustrations | core | refers to any processing element capable of (mostly) computing the target hash without having a lot of memory of its own. We give memory to cores, sometimes sharing it.

A core may be today's usual CPU core, or it may be a SIMD (single instruction, multiple data) unit (e.g. within a GPU CU or SM), or it may be a SIMD lane (within a CPU core or a GPU SIMD unit), or it may be a pipeline stage (e.g. with different hash computations' instructions interleaved on a superscalar CPU or a GPU), or it may even be a single bit number across N-bit registers in a register file (when we've rotated our problem and are now "bitslicing" it)

Of course, and most importantly, a core may also be a logic circuit in an FPGA or ASIC, but even there by a core we might also be referring e.g. to each pipeline stage, whichever option is relevant or optimal in a given context

Bitslicing

* Eli Biham, "A Fast New DES Implementation in Software", 1997
  ~100 logic gates per S-box

"about three times faster than our new optimized DES implementation on 64-bit computers. [...] view the processor as a SIMD computer, i.e., as 64 parallel one-bit processors computing the same instruction." (It's commonly 512 now.)

* Matthew Kwan, "Reducing the Gate Count of Bitslice DES", 1997+
  87.75 (1997), 51 to 56 (1998) on average depending on available gate types

* Marc Bevand (45.5 with bit select), Dango-Chu (39.875, ditto), Roman Rusakov (32.875 with bit select, 44.125 without), DeepLearningJohnDoe (23.5 with LUT3), Sovyn Y. (22.125 with LUT3 - later in 2024, not in crackers yet)

* On AVX-512, bitslicing lets us eliminate 512 computed hashes in ~9 steps

Wrapping up on hashing speed optimization

* Current most flexible password crackers - John the Ripper and hashcat - support hundreds of different (non-)hash types on many hardware platforms

* It takes a lot of effort to optimize each combination, so the extent of optimization at a given time varies across tools, (non-)hashes, platforms

* Where practical, we find ways to share optimizations across (non-)hash types and/or platforms - e.g. John the Ripper's shared SIMD "pseudo intrinsics"

* Related higher-level efforts include Aleksey Cherepanov's john-devkit and Alain Espinosa's fast-small-crypto, which are specialized code generators

* OpenCL abstracts much of this as well, so recent hashcat uses it even on CPU

* Tuning, including runtime on the target machine (e.g. OpenCL work sizes)

Speeds for historical Unix hashes

* Unix up to 6th Edition, based on M-209

c/s	year	software	hardware	power
800	mid 1970s		PDP-11/70	

* Unix 7th Edition, 25 iterations of salt-modified DES, amortized multi-salt

3.6	1977		VAX-11/780	
45	1988	Morris worm	VAX 6800	
<= 1k	~1993	Crack, Cracker Jack	386DX 40 MHz with cache	
12.5k	1998	John the Ripper 1.5	Pentium 133 MHz	
214k	1998	John the Ripper 1.5	Alpha 21164A 600 MHz	
973M	2019	John the Ripper 1.9	ZTEX 1.15y FPGA (2012)	34W
6277M	2022	hashcat 6.2.6	RTX 4090 GPU	~450W
1825M	2024	John the Ripper	2x Xeon Platinum 8488C	~770W

Speeds for contemporary decent hashes

bcrypt at 32 iterations (cost 5), even though modern uses are at 256 to 4096 (cost 8 to 12), so would be ~8 to ~128 times slower

c/s	year	software	hardware	power
6.5	1998	OpenBSD 2.3 library	Pentium 133 MHz	
22.5	1998	John the Ripper 1.5	Pentium 133 MHz	
62.5	1998	John the Ripper 1.5	Alpha 21164A 600 MHz	
6595	2013	John the Ripper 1.8	i7-4770K ~3.7 GHz turbo	~84W
185k	2017	hashcat 3.5.0-22-..	8x GTX 1080 Ti GPU	
106k	2017	John the Ripper	ZTEX 1.15y FPGA 141 MHz	
119k	2019	John the Ripper 1.9	ZTEX 1.15y FPGA 150 MHz	27W
2.1M	2019	John the Ripper 1.9	18x ZTEX 1.15y + host	585W
~25k	2019	hashcat, JtR	Vega 64 GPU	
<250k	2022	hashcat 6.2.6	RTX 4090 GPU	<450W
169k	2024	John the Ripper	2x Xeon Platinum 8488C	~770W

Password cracking: past, present, future

Speeds for historical Windows hashes

L0phtCrack 1.5 "on a Pentium Pro 200 checked a password file with 10 passwords using the alpha character set (A-Z) in 26 hours. [...] [note from mudge: try building the CLI version on an ultrasparc using the compile flags in the Makefile provided - this will make these figures look slooooowwww ;-)]" (1997)

* This means 89k p/s against 10 LM hashes, which JtR improved upon shortly

* Hash Suite 3.5 by Alain Espinosa: 8360M on HD 7970, 7270M on GTX 970 (2018)

* hashcat 4.1: 4134M on HD 7970, 5250M on GTX 970 (2018) (all at 10 hashes)

* hashcat 6.2.6 vs. one LM hash (10 would be slightly slower), RTX 4090 (2022)

```
Speed.#1.........:   151.1 GH/s (6.95ms) @ Accel:256 Loops:1024 Thr:32 Vec:1
```

Speeds for contemporary Windows hashes

* hashcat 6.2.6 vs. NTLM, RTX 4090

```
Speed.#1.........:   288.5 GH/s (7.24ms) @ Accel:512 Loops:1024 Thr:32 Vec:8
```

* hashcat 6.2.6 vs. NTLM, 8x RTX 4090 (different rig, driver, CUDA version)

```
Speed.#1.........:  258.6 GH/s (16.22ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#2.........:  255.1 GH/s (16.40ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#3.........:  255.3 GH/s (16.43ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#4.........:  257.4 GH/s (16.26ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#5.........:  251.7 GH/s (16.64ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#6.........:  230.3 GH/s (18.27ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#7.........:  228.7 GH/s (18.37ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#8.........:  256.7 GH/s (16.35ms) @ Accel:128 Loops:1024 Thr:256 Vec:1
Speed.#*.........:  1993.9 GH/s
```

Password cracking: past, present, future

Speeds for contemporary strong hashes

```
* John the Ripper vs. Argon2, GTX 1080 (at max turbo, 180W, 2016 gaming GPU)

Cost 1 (t) is 3 for all loaded hashes
Cost 2 (m) is 16384 for all loaded hashes
Cost 3 (p) is 1 for all loaded hashes
Cost 4 (type [0:Argon2d 1:Argon2i 2:Argon2id]) is 0 for all loaded hashes
Trying to compute 480 hashes at a time using 7680 of 8119 MiB device memory
LWS=[32-256] GWS=[15360-15360] ([60-480] blocks) => Mode: WARP_SHUFFLE
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
03091983         (?)
1g 0:00:00:28 0.03457g/s 1742p/s 1742c/s 1742C/s Dev#4:53C thriller1..d1amond

* 2x Xeon E5-2670 + 8x DDR3-1600 (32 threads, 230W total, 2012 server CPUs)

1g 0:00:01:54 0.008726g/s 437.8p/s 437.8c/s 437.8C/s 050489..010591
```

Wrapping up on speeds

* So far we discussed how to optimize the speed
  + of hashing
  + of matching against loaded hashes

* We mostly didn't yet discuss optimizing the speed
  + of candidate password generation
  + of duplicate candidate password avoidance or suppression

* For contemporary strong targets the speeds per device haven't obviously changed much since the 1970s and are mostly within one or two orders of magnitude of 1000 checks per second

* At such low speeds, other optimization goals matter most (focus, targeting, uniqueness vs. memory requirements, ease of reasoning, feedback, workflow)
* Amdahl's law has effect when other processing is sequential and synchronous

PASSWORD CRACKING FOCUS

```
/*
 * Warning: this program burns a lot of cpu.
 */
/*
 * Insecure - find accounts with poor passwords
```

Date: Tue, 29 Nov 83 18:19:32 pst
From: leres%ucbarpa@Berkeley (Craig Leres)

Insecure is something that Jef Poskanzer and I wrote to rid a local system of an overly persistent ankle-biting adolescent. It was a quick hack we whipped up in just a few minutes and was never intended to be publically distributed. Unfortunately, I made the mistake of giving a copy to an associate at UC Berkeley. Apparently, he incorporated it in a security package he later developed for use at Berkeley. Someone else distributed it outside Berkeley which explains why it's been publically distributed.

Candidate password generators (1980s)

```
static char *rcsid = "$Header: pwchkr.c,v 1.1 85/09/10 16:00:56 root Exp $";

 * By default, this program only checks for accounts with passwords the same
 * as the login name. The following options add more extensive checking.

 *              -w file: use the list of words contained in "file" as likely
 *                       passwords. Words in the file are one to a line.
 *              -b:      check all guesses backwards too
 *              -g:      use the Full Name portion of the gecos field to
 *                       generate more guesses; also check .plan, .signature
 *                       and .project files.
 *              -s:      check the single letters a-z, A-Z, 0-9 as passwords
 *              -c:      with each guess, check for all-lowercase and
 *                       all-uppercase versions too.
 *              -d:      check the doubling of the username
```

Candidate password generators (early 1990s)

"The first pass that Crack makes is over the [data] gleaned from the users' password field. In my test file, this gets about 4% of the passwords (out of a total of 15% guessed). This pass also completes very quickly, working as it does from a very small amount of data, but one which is very frequently used for passwords.

The first sweep of the second pass, consisting of lowercase dictionary words, gets about another 5% of the passwords. The length of the first sweep depends on how much CPU and how many dictionaries I supply, but using the Ultrix /usr/dict/words and my bad_pws.dat, over 4 CPUs, it doesn't take much more that a few hours.

For the further sweeps, the percentages cracked per sweep tail off, 2%, 1%, 0.5%... But they are the people with fairly exotic passwords, and it's only common sense that that they will take some time to crack." - Alec Muffett

Candidate password generators (mid 1990s)

Excerpts from Crack user manual by Alec Muffett:

"Crack 5.0 supports the notion of dictionary groups - collations of words taken from a selection of raw text dictionaries (with words given, one per line) permitting the user to group her dictionaries into "most-likely", "less-likely" and "least-likely" to generate a successful password guess."

"When Crack starts up [...] two other dictionary groups are created: "gecos" and "gcperm". The "gecos" group contains only words directly derived from the information held in the password file; the "gcperm" group holds words which are mechanically created by permuting and combining parts of words held in the password file (eg: "Alec Muffett" becomes "AMuffett", "AlecM", etc)."

"When the cracker is running, it [is] taking successive mangling rules [...] and applying them to the cited dictionary group"

Mangling rules

"These rules are macro commands, one per line, which specify patterns and actions that are applied to words from a dictionary in order to generate a series of guesses.

For instance, one such rule:

```
/ese3u
```

...will select words which contain the letter "e", replace it with the digit "3", and force the rest of the word to uppercase." (Can also be "/e se3 u".)

* Introduced in Crack 4.0 (Nov 1991), maintained until Crack 5.0 (Dec 1996)

* Adopted and extended in John the Ripper (1996+), InsidePro's tools, hashcat

Mangling rules evolution

* Specific to John the Ripper (not adopted by InsidePro and hashcat)
  + Preprocessor
  + Rule reject flags (e.g. skip a rule if hash is case-insensitive)
  + Word pair commands (used on concatenated first and last names only)

```
# johnsmith -> John Smith, John_Smith, John-Smith
-p-c 1 <- (?a c $[ _\-] 2 (?a c
```

* Other extra commands in John the Ripper
* Same and different extra commands in InsidePro's PasswordsPro & Hash Manager
* Same and different extra commands in hashcat (started with PasswordsPro's)

* hashcat "World's first and only in-kernel rule engine" (OpenCL and CUDA)

* hashcat compatibility mode in John the Ripper

Mangling rulesets

* Old John the Ripper default rules were hand-written, some tuned, some not

* KoreLogic rules targeting users' coping with password policies (2010)
  + Rewritten to use the preprocessor (much shorter, expands to ~7M rules)

* InsidePro's PasswordsPro rules imported by the hashcat community, many new rulesets created by the community over the years (hand-written, generated)
  + hashcat best64 contest - come up with most efficient 64 rules

* OneRuleToRuleThemAll - best 25% of all hashcat rules (~52k rules)
  + OneRuleToRuleThemStill - optimized further (~49k rules)

* Passphrase rules for hashcat and John the Ripper (some expect special input)

* John the Ripper "All" ruleset (using nested include directives) is ~11M

Mangling ruleset tuning

* Simon Marechal, "Automatic mangling rules generation", 2012 specifically for John the Ripper, but didn't result in anything integrated in the project
  + Original implementation of rulesfinder abandoned, project re-born in 2020

* John the Ripper PerRuleStats feature added and used to re-order ~7600 line preprocessor output from certain hand-written rules, now new defaults (2022)

"[...] decreasing weighted score, which considers number of guesses both per rule and per password candidates tested. This ordering is good to use on fast to medium speed hashes. Generated [...] on pwned-passwords-sampler output for HIBP v8 (100M non-unique, ~54M unique hashes), counting unique guesses.

Manually split into best (40%), worse (next 40%), and worst (final 20%), which achieve, respectively, most of the cracks (97%+), some more (2%+), and hardly any."

Targeting

* When deriving candidate passwords from users' information, check those only against the same users' hashes
  + ... and against other hashes with the same salt since it's almost free

* Cracker Jack's inconvenient "single crack" mode (1993)
  + Run a separate program to create a custom format wordlist, then use it

* John the Ripper's convenient "single crack" mode (1996)
  + Everything built-in and automatic
  + Also checks successfully cracked passwords against all hashes

* Markus Duermuth et al. OMEN+ Markov model with personal information (2013)

* hashcat's Association mode (2020)
  + changes.txt: [...] attack hashes from a hashlist with associated "hints"

Duplicate candidate passwords

Consider the following entries on a common passwords list used as a wordlist:

password
password1
Password

Then the rule "l $1" (lowercase and append the digit 1) will produce:

password1
password11
password1

This has a duplicate, and besides "password1" is duplicate with the original wordlist (the output of a likely preceding no-op rule)

Duplicate candidate password avoidance

* Crack's rules support many "reject the word if/unless ..." commands, which can be used to avoid producing most effectively-duplicates

* John the Ripper added more word reject commands and also rule reject flags, and its default ruleset makes extensive use of all of these (not trivial)

* With input wordlists constrained to lowercase-only, this is very effective; with common passwords as input, not too bad (e.g. 89.6% unique for default wordlist and ~3k rules, which the duplicate candidate password suppressor at 256 MiB RAM usage improves to 94.3%, producing a ~50 GB output stream)

* hashcat rulesets typically do not use rejects much or at all, resulting in many more duplicates (e.g. 59.7% unique for the same wordlist as above with top 3k rules from OneRuleToRuleThemStill in John the Ripper's compatibility mode, which the 256 MiB RAM duplicate suppressor improves to 80.6%)

Duplicate candidate password suppression

* John the Ripper "single crack" uses small per-salt ring buffers (along with hash tables for fast lookup) to detect and suppress recently seen candidates

* hashcat "Supports password candidate brain functionality" (2018)
  + Network client/server architecture - can suppress duplicates across jobs
  + Requires (easy) manual setup even for local single-job use
  + Per documentation, server has performance bottleneck at 50k p/s
  + Writes fast hashes of password candidates to disk, no limit, no eviction

* John the Ripper duplicate candidate password suppressor (2022)
  + Currently per-process - won't suppress duplicates across processes
  + Enabled by default when mangling rules are in use, auto-disables when hit rate is low and other processing speed is high, memory use can be tweaked
  + Speed of a few million p/s, but sequential and synchronous with the rest
  + Opportunistic - probabilistic filter of pre-specified size, has eviction

Duplicate candidate password suppressor

John the Ripper's duplicate candidate password suppressor uses a hash table to store fingerprints (other fast hashes) of items. This is similar to a cuckoo filter, except that it's degraded from having 2 to only 1 potential bucket for each item (so not cuckoo). The buckets are currently 8 items wide. When the bucket is full, we simply evict/replace a fingerprint (from the second half).

```
(index, fp) = two_fast_hashes(candidate)
   |    +--------------------+
   |        LOCKED ADD-ONLY  v   |         EVICTABLE
   | 0------1------2------3------4------5------6------7------+
   | |  fp  |  fp  |  fp  |  fp  |  fp  |  fp  |      |      |
   | 1------+------+------+------+------+------+------+------+
   +>|  fp  |  fp  |  fp  |      |      |      |      |      |
     2------+------+------+------+------+------+------+------+
     |  fp  |  fp  |  fp  |  fp  |  fp  |  fp  |  fp  |  fp  |
```

Wordlist de-duplication tools

The task is to remove duplicate lines without changing the order (which may have been optimized in some way), so without sorting, and without requiring memory for the whole output nor disk storage beyond that for the output

* John the Ripper 1.6+ bundled "unique" (1998)
  + When output exceeds memory, re-reads the output file written so far to identify duplicates between what's in memory and what's already output

* duplicut (2014)
  + Multiple threads "are only used when the file is huge enough to chunk"

* hashcat-utils rli (2015)

* Rling, "a faster multi-threaded, feature rich alternative to rli" (2020)
  + by Waffle (author of MDXfind password cracker)

Wordlists

* Historically, password crackers literally used lists of dictionary words

* Tiny public common password lists appeared in 1980s, e.g. from Morris worm

* Moderately longer ordered list was maintained in John the Ripper (1997+)

* RockYou leak (32.6M plaintext passwords, 14.3M unique) changed a lot (2009)
  + A large sample of passwords that are not biased to what was crackable
  + The standard for password security tools' training, testing, and usage

* Lists of dictionary words were and still are used as well (e.g. Openwall wordlists collection, 2003, which merges and credits lots of sources)

* Additional published wordlists were scraped e.g. from Wikipedia (Sebastien Raveau, 2009, 2012) and Project Gutenberg books (CrackStation, 2010)

Password cracking communities

* Hobbyist community forums such as InsidePro's, hashkiller.co.uk, hashes.org (all of which are now defunct) collected uploads of hashes to crack and the plaintexts members cracked so far, typically without identifying the source and without the usernames

* Files from the above were also retrieved by the defensive security community

* Have I Been Pwned (HIBP) or "Pwned Passwords are hundreds of millions of real world passwords previously exposed in data breaches", published by Troy Hunt in the form of SHA-1 and NTLM hashes (re-hashed from plaintexts) along with numbers of occurrences in breaches
  + passwdqc is able to proactively check new passwords against HIBP offline

* It is still possible to obtain the hashes.org plaintext lists elsewhere, re-crack almost all of HIBP and generate an ordered breached passwords list

Wordlist optimization

HIBP v8 being at 847M unique passwords (from a few billion accounts) is large (although perhaps not the largest collection in existence).

RockYou is arguably cleaner. Both are fair play for security researchers.

John the Ripper's current password.lst (2022, generated in time for the mangling ruleset tuning) is "based on Pwned Passwords v8 (HIBP) 100+ hits overlap with RockYou, further filtered to require 97+ hits on top of RockYou's. These criteria are such that a password used by just one person many times is very unlikely to be included."

* A focused common passwords list of ~1.8M lines and ~15 MB

* Obviously ethical to redistribute and highly effective

Probabilistic candidate password generators (mid 1990s)

* Probabilistic password generator is a "technique for generating candidate passwords from a statistical model" (Simon Marechal, 2012)

* Novel algorithm to search the keyspace exhaustively and without duplicates while walking the 2D surface of Charset ** Length uphill (1995)

* John the Ripper 1.0 introduced "incremental mode" (1996)

```
[Incremental:Alpha]
CharCount = 26
MinLen = 1
MaxLen = 8
CharsetB = smcbtdpajrhflgkwneiovyzuqx
CharsetM = eaiornltsuchmdgpkbyvwfzxjq
CharsetE = erynsatldoghikmcwpfubzjxvq
```

Probabilistic candidate password generators (later 1990s)

* John the Ripper 1.0's may retroactively be called a 0th-order Markov chain

* A further 0th-order variation added per-length and per-position statistics

```
Charset11 = ajyfioqxdehmnrstlcupbgkwvz

Charset21 = mdjpagetbrnsckyfilwhuoqvzx
Charset22 = olstabegrkjdhnvwcmpfiquxyz

Charset31 = dacjmbrtpslknfeghowqvzxiuy
Charset32 = aoeisumctgdblrfjpnvhwkxyzq
Charset33 = msnctdxepghlywabrjikuzofvq
```

* Star Cracker by The SOrCErEr went beyond 0th-order (late 1996)
* John the Ripper 2nd-order along with per-length and per-position (late 1996)

Probabilistic candidate password generators (late 1990s)

* Training on previously cracked passwords (reading john.pot) as a feature in John the Ripper (obscure releases in late 1996, popular 1.4 in early 1997)

```
-makechars:<file>      make a charset, <file> will be overwritten
```

* John the Ripper 1.5 upgraded the surface to walk uphill on from 2D to 3D and the height from keyspace portion to expected cracks per candidate (1998)

* The 3 components are: current virtual character count, current length, and current position of fixed virtual character
  + The 3rd component was internal to the earlier algorithm, now exposed
  + "Virtual character" means its index into a sorted string, which vary depending on up to 2 preceding characters (2nd-order Markov chain)

* The finer granularity is later helpful for parallel/distributed processing

Vowel/consonant patterns

* John the Ripper 1.0 to 1.4 had

```
Wordlike   Set to 'Y' to enable a simple built in word filter (words with
           more than one vowel in a row, or more than two non-vowels in a
           row, will get filtered out).
```

This was needed along with 1.0's 0th-order Markov (non-)chain, which was still available as an option in 1.4 for memory saving (2nd-order required 4 MB RAM) and for possible manual specification of character sets per position (similar to future "mask mode").

* John the Ripper 1.5 no longer had 0th-order at all, so the above was dropped

* Specifically exploiting vowel/consonant patterns is a recurring idea

Custom candidate password generators (late 1990s)

* John the Ripper 1.3+ "external mode" (late 1996 or early 1997)
  + Write your own candidate password generator or filter in a C-like language in the configuration file
  + Compiles to stack-based VM implemented via threaded code
  + Optimizations: top of stack cache, multi-push, GCC "Labels as Values"
  + Optional JIT to 32-bit x86 (1997), abandoned in the rewrite for 1.5 (1998)

* Incremental mode training filters (initial use case)
* Password policy matching filters

* Recovery of partially lost passwords (mostly obsoleted by mask mode)
* Exploits for Strip, DokuWiki, KDEPaste, Awesome Password Generator
* Date/time, keyboard walks and other sequences, any same-character repeats, passwords with any few different characters (now built-in Subsets mode)
* Searches of short valid Unicode strings (auto-generated from Unicode spec)

Custom candidate password patterns (2000s+)

Crackers that didn't have probabilistic candidate password generators instead added features to focus dumber exhaustive searches on reasonable sub-spaces

* InsidePro introduced the mask syntax e.g. ?u?l?l?l20?d?d
  + Reuses the character class notation from Crack's word reject rules
  + Enumerates all strings matching the mask e.g. Aaaa2000 to Zzzz2099

* hashcat adopted the syntax, later extended the implementation to use Markov

* John the Ripper later also adopted this syntax as mask mode, extending it with constructs similar to the rule preprocessor's e.g. ?u?l?l?l20[0-2]?d

* Hybrid modes add a mask on top of a smarter and slower generator (2010s)
  + hashcat modes append or prepend a mask (on device) to wordlist (from host)
  + John the Ripper (on device) mask ?w refers to another (host) mode's "word"

Probabilistic candidate password generators (early 2000s)

* Dawn Xiaodong Song, David Wagner, and Xuqing Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH", 2001

"for passwords that are chosen uniformly at random with length of 7 to 8 characters, [...] can reduce the cost of password cracking by a factor of 50"

"we model the relationship of latencies and character sequences as a Hidden Markov Model. We extend the standard Viterbi algorithm to an n-Viterbi algorithm that outputs the n most likely candidate character sequences."

* Code never released, third-party n-Viterbi implementations appeared later

* Independently, our "SSH Traffic Analysis" (with Dug Song, 2001) allowed to infer sudo, etc. password lengths from packet traces, and countermeasures for that were added, but it took until OpenSSH 9.5 (2023) for the timings

Probabilistic candidate password generators (mid 2000s)

* Arvind Narayanan and Vitaly Shmatikov, "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff", 2005

"Our first insight is that the distribution of letters in easy-to-remember passwords is likely to be similar to the distribution of letters in the users' native language. Using standard Markov modeling techniques from natural language processing, this can be used to dramatically reduce the size of the password space to be searched. Our second contribution is an algorithm for efficient enumeration of the remaining password space. This allows application of time-space tradeoff techniques, limiting memory accesses to a relatively small table of "partial dictionary" sizes and enabling a very fast dictionary attack."

"We note the similarity of the ideas used in this algorithm to the well-known Viterbi algorithm from speech processing"

Probabilistic candidate password generators (circa 2010)

* Simon Marechal, "Etat de l’art sur le cassage de mots de passe", 2007
  + Narayanan and Shmatikov's work re-applied to classical password cracking

* John the Ripper jumbo "Markov mode" contributed by Simon Marechal, extended by Frank Dittrich and magnum (2007-2012+)

* 1st-order Markov chain, no per-length and per-position separation

* Outperforms "incremental mode" on certain tests, but requires advance choice of attack duration (via minimum and maximum strength of passwords)

* Supports parallel and distributed processing (limiting a node's sub-range)

* Simon Marechal, "Probabilistic password generators", 2012
  + Comparison of many probabilistic models (including 2nd-order variations)

Probabilistic candidate password generators (2010s)

* Matt Weir et al., "Password Cracking Using Probabilistic Context-Free Grammars", 2009 or Pretty Cool Fuzzy Guesser (PCFG)

"new method that generates password structures in highest probability order. We first automatically create a probabilistic context-free grammar based upon a training set of previously disclosed passwords. This grammar then allows us to generate word-mangling rules, and from them, password guesses"

* Markus Duermuth et al., "OMEN: Faster Password Guessing Using an Ordered Markov Enumerator", 2013

"Narayanan et al.'s indexing [not] in order of decreasing probability. [We] enumerate passwords with (approximately) decreasing probabilities. On a high level, our algorithm discretizes all probabilities into a number of bins, and iterates over all those bins in order of decreasing likelihood." (3rd-order)

Candidate passphrase generators (mostly 2010s)

* Wordlist rules appending/prepending specific embedded words

* Trivial word-combining Perl scripts posted to john-users (2006)

* hashcat Combinator mode (2 words from 2 lists, not probabilistic)

* PRINCE (PRobability INfinite Chained Elements) by atom (2014)
  + Sorts for increasing combined length, otherwise not probabilistic
  + hashcat project's princeprocessor
  + Kindly also contributed to John the Ripper, became a built-in mode

* Passphrase mangling rulesets like for wordlists, but expect phrases (2019)

* Passphrase lists e.g. extract all 2 to 6 sequences from Project Gutenberg books, sort from most to least common (2021, unreleased)

Probabilistic candidate passphrase generators (2010s+)

* Probabilistic candidate password generators also happen to generate phrases if trained on such input (or just on a real-world mix of passwords/phrases)
  + PCFG fares better than per-character Markov chains

* "Phraser is a phrase generator using n-grams and Markov chains to generate phrases for passphrase cracking" in C# for Windows (2015)

* RePhraser "Python-based reimagining of Phraser using Markov-chains for linguistically-correct password cracking" (2020)
  + Also includes related hand-written and generated rulesets

* What about middle ground (e.g. syllables, including some followed by space)?
  + e.g. extract all substrings of 2+ characters, sort from most to least common, take top ~100, map them onto indices along with single characters, train/use existing probabilistic candidate password generators, map back

Probabilistic candidate password generation with neural networks (2010s+)

* William Melicher et al., "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks", 2016
  + Recurrent neural network (RNN) predicts next character, no duplicates
  + 60 MB model outperforms other generators, but apparently was too slow to actually go beyond 10 million candidates so that is only simulated
  + 3 MB performs almost as well, takes ~100 ms per password in JavaScript

* Generative Adversarial Networks (GAN) produce duplicates (~50% at 1 billion)
  + "PassGAN: A Deep Learning Approach for Password Guessing" (2017)
  + "Improving Password Guessing via Representation Learning" (2019)
  + "Generative Deep Learning Techniques for Password Generation" (2020)
    + David Biesner et al., VAE, WAE, fine-tuned GPT2 - maybe currently best?
  + "GNPassGAN: Improved Generative Adversarial Networks For Trawling Offline Password Guessing" "guessing 88.03% more passwords and generating 31.69% fewer duplicates" than PassGAN, which had already been outperformed (2022)

Candidate password generator combinations

* Different generators produce some unique and some overlapping candidates
  + It is desirable to use multiple generators and suppress cross-duplicates
  + In practice so far, it is most common to use multiple, but not suppress
  + John the Ripper's default invocation does suppress duplicates between its wordlist and incremental mode passes; hashcat's brain may do similar, too
  + Another way to suppress is by exhausting easy patterns (or planning to), then excluding them from complex runs (similar to policy matching)
    + Maybe even at the expense of most optimal order for some runs

* Generators may be optimized for usage along with some other ones
  + e.g. probabilistic ones are generally to be used after some wordlist runs
  + training a probabilistic generator to perform best on its own means it may be overfit (mimic a wordlist) and crack fewer passwords beyond wordlist

* Comparisons often fail to take this into account, but they should

Workflow

* Unfortunately, mostly did not fit in this talk, but is very important

* Best results are achieved by using multiple approaches in multiple steps
  + Beyond usage of multiple candidate password generators
  + Cracking progress so far should inform further actions
  + Cracked passwords so far should be used as wordlist and for re-training

* Workflow automation / job management tools exist
  + Functionality overlaps with distributed processing and team coordination
    + e.g. in contests and red teams' work
  + Crack had some of this bundled
    + its scripts split the workload and could run jobs over rsh
  + Tools like Hashtopolis (nee Hashtopussy) by s3inlc take this much farther

* Writeups from contests starting with KoreLogic's Crack Me If You Can (2010+)

Future

* Speed
  + Obvious: larger and higher-clocked CPUs, GPUs, FPGAs
  + Major: publicly available ASICs, open hardware FPGAs (or both on one die)
  + Minor: further optimizations (e.g. bitsliced Lotus/Domino coming soon)

* Focus
  + Better passphrase support (tools, datasets), arbitrary tokenization
  + Further neural networks, tackling the duplicates problem of generative NNs
    + Meanwhile, publicly release pre-generated and pre-filtered output
  + Application of NNs for targeting (scraping and training on user data)

* Features
  + More (non-)hashes; yescrypt ROM support; usage of NVIDIA unified memory
  + Easy distributed processing (easy setup, dynamic re-assignment, security)
  + Ease of use, including by one-time end-users - UI or LLM guiding the user

Takeaways

Password cracking is

* simple on the surface, with low barrier to entry and gradual learning curve

but it also is

* serious computer science and engineering, with non-trivial social aspects
* still an evolving and highly competitive field welcoming new contributors

* Efficiency can mean win or lose, rich or poor

* Like with other offensive security fields, new techniques and results inform design and parameters of new defenses

* Public availability helps level the playing field

Contact information and credits

e-mail
Solar Designer <solar@openwall.com>

website
https://www.openwall.com

Twitter
@solardiz @Openwall

Thanks to CIQ, the primary corporate sponsor of Rocky Linux, for encouraging me to give this talk https://ciq.com