Key Generation and Derivation

Levente Buttyán

CrySyS Lab, BME

buttyan@crysys.hu

Outline

- Key generation
  - symmetric keys --» random number generation
  - asymmetric keys --» randomized algorithms --» rand num generation
- Key derivation
  - from high entropy input
  - from low entropy input (e.g., password)
- Password authenticated key exchange (PAKE)

Random number generation

The need for random numbers

- Random numbers (bits) are needed for various purposes, including for forming cryptographic keys (both symmetric and asymmetric) and other cryptographic parameters (e.g., unpredictable IVs, nonces, etc.)

- Random number generators available in standard programming libraries are not good for cryptographic purposes
  - example: congruential generator
    - output $s_{i+1} = (a \cdot s_i + b) \bmod n$
    - has nice statistical properties
    - but it is predictable!

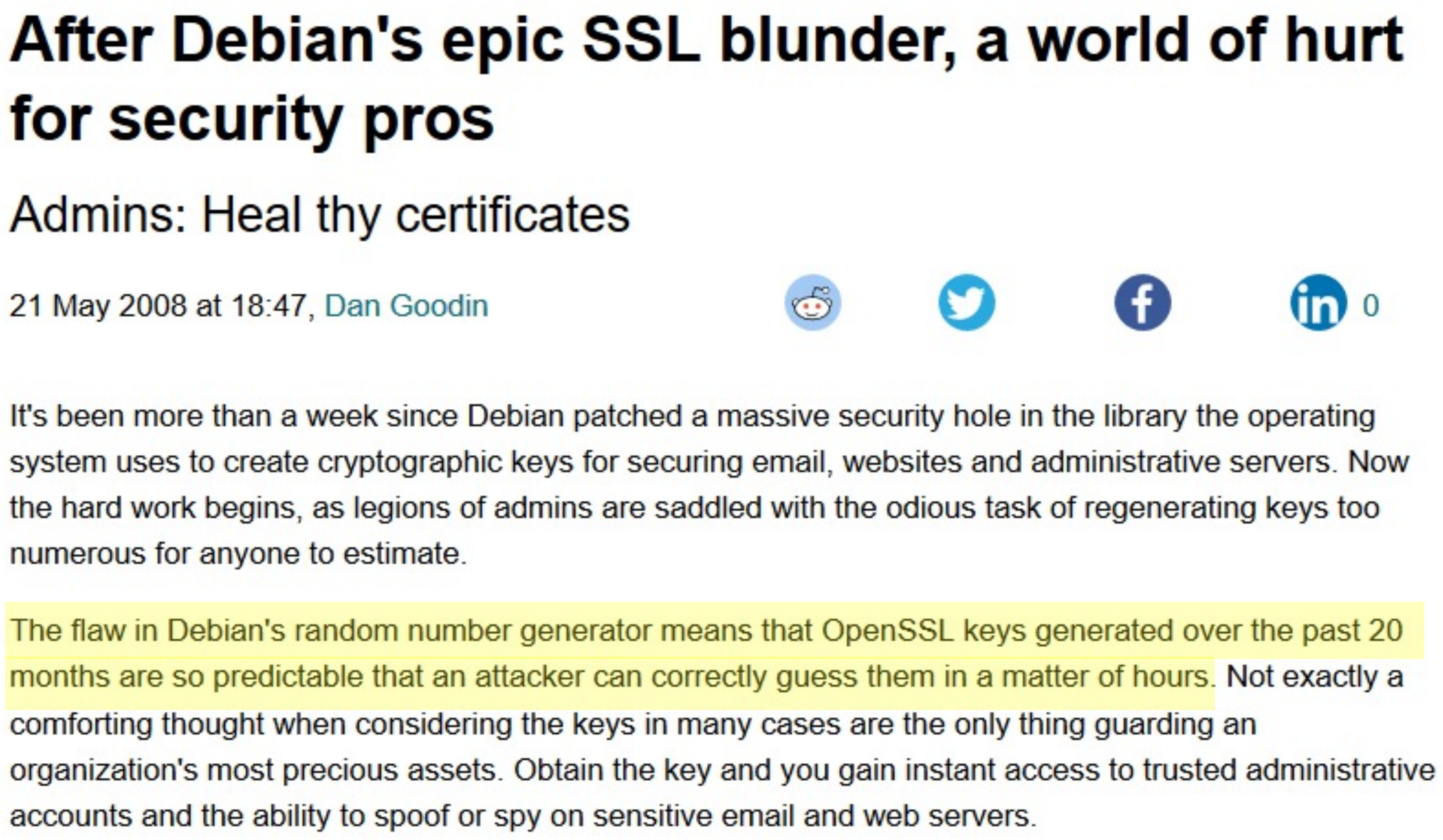
- Weak random number generators can easily destroy security even if strong cryptographic primitives (ciphers, MACs, etc.) are used

Random number bug in Debian Linux

- Introduced in 2006, discovered in 2008
- Someone commented out a single line of code in the OpenSSL package of Debian Linux, because some code verification tools produced a warning on it

```
/*
 * Don't add uninitialised data.
       MD_Update(&m,buf,j); /* purify complains */
*/
```

- This crippled the seeding process of the OpenSSL random number generator
  - the only ”random” value that was used to initialize the generator was the current process ID
  - process IDs on Linux can take 32768 possible values
  - very small number of possibilities to try by an attacker!



After Debian's epic SSL blunder, a world of hurt for security pros

Admins: Heal thy certificates

21 May 2008 at 18:47, Dan Goodin

0

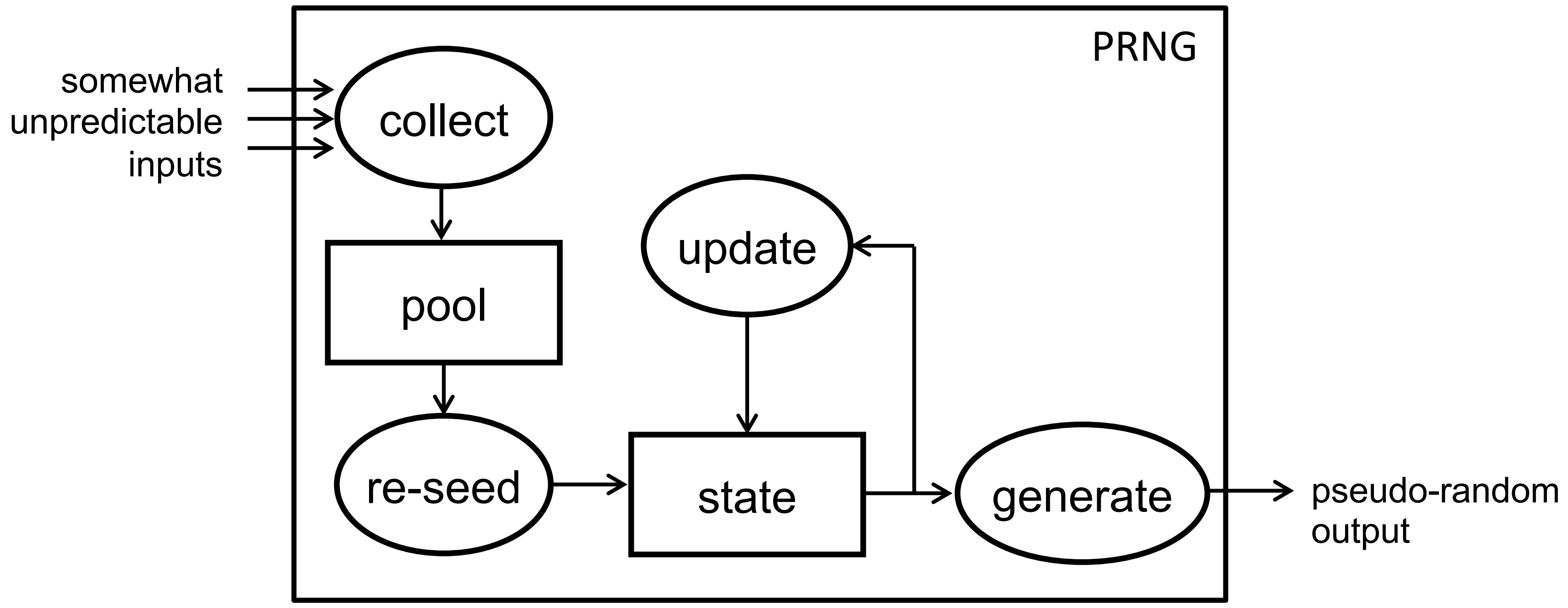
It's been more than a week since Debian patched a massive security hole in the library the operating system uses to create cryptographic keys for securing email, websites and administrative servers. Now the hard work begins, as legions of admins are saddled with the odious task of regenerating keys too numerous for anyone to estimate.

The flaw in Debian's random number generator means that OpenSSL keys generated over the past 20 months are so predictable that an attacker can correctly guess them in a matter of hours. Not exactly a comforting thought when considering the keys in many cases are the only thing guarding an organization's most precious assets. Obtain the key and you gain instant access to trusted administrative accounts and the ability to spoof or spy on sensitive email and web servers.

True random vs. pseudo-random numbers

- A **true random number** is a number that cannot be predicted by an observer before it is generated
  - if an integer is generated within the range [1, N], then its value cannot be predicted with any better probability than 1/N
  - this is true even if the observer has unlimited computing power and he is given all previously generated numbers
- True random numbers are not always available in sufficient quantity or fast enough
- A cryptographic **pseudo-random number generator (PRNG)** is a mechanism that processes somewhat unpredictable inputs and generates pseudo-random outputs (fast and in large quantity)
  - if designed, implemented, and used properly, then a **computationally bounded adversary** cannot distinguish the PRNG output from a real random sequence

General structure of PRNGs



- state is
  - updated after every output
  - recomputed when sufficient amount of entropy is collected from the inputs
- generation is typically based on some one-way function
  - hash or block cipher

“Somewhat unpredictable inputs”

- Possible sources:
  - current time
  - keystroke timings
  - mouse movement
  - disk access times
  - camera image, microphone
  - special hardware
    (e.g., thermal noise of resistors)

- Collected bits are not necessarily all unpredictable and independent, and the adversary might know or even influence some part of them

- What is important is that the harvested bits contain some information (entropy) which is unguessable to the adversary

The Fortuna PRNG

- Design principles:
  - Accumulate entropy from as many different sources as possible
  - Re-seed (re-generate state) occasionally and ensure that at some point in time the PRNG is put in a surely unguessable state
  - Between re-seeds, use strong crypto algorithms to generate outputs from the internal state
- Main components:
  - Output generator
  - Re-seed control
  - Accumulator (entropy collector)

Output generator

- Based on a 128-bit block cipher (e.g., AES or TwoFish) used in CTR mode with a 256-bit key

- Internal state
  - C – 128-bit counter
  - K – 256-bit key

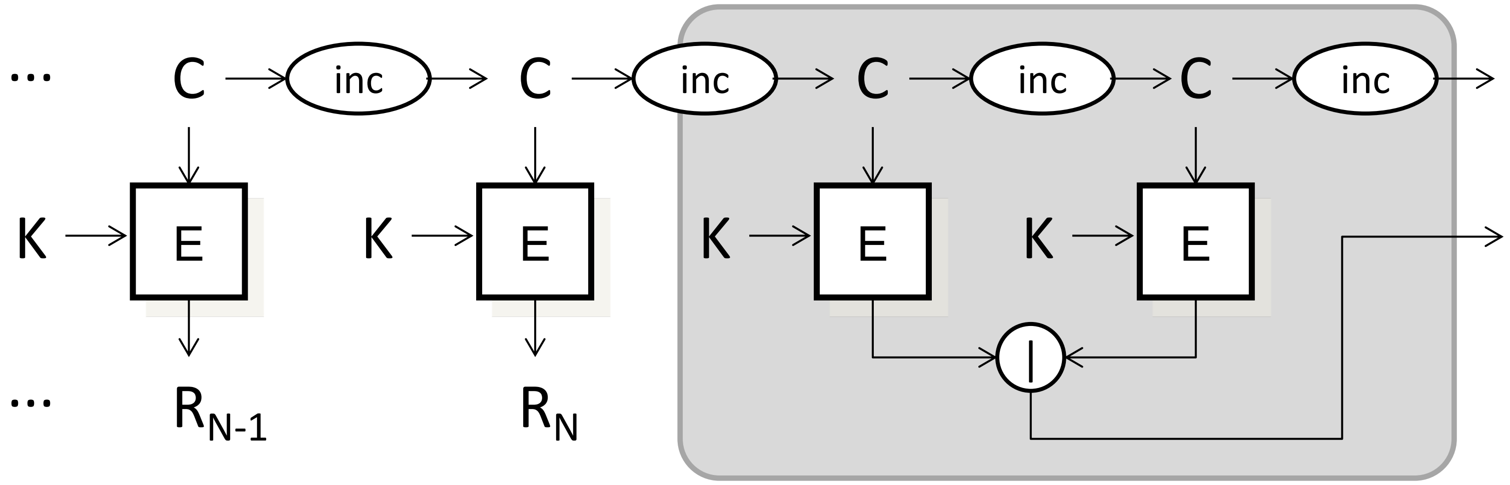
- Generation algorithm:
  - input: number N of blocks to generate
  - output: N random blocks (i.e., 16*N bytes)

```
R = empty string
for i = 1, 2, ..., N:
        R = R | $E_K(C)$
        C = C+1
return R
```

Usage of the output generator

- Call the generation algorithm with an appropriate input N to generate the required amount of random bytes
  - if L random bytes are needed, then
    - call the generator with N = ceil(L/16)
    - take the first L bytes of the generated output

- Switch to a new key to avoid later compromise of the output just generated
  - call the generator with N = 2
  - set the new value of K to the generated 2*16 = 32-byte (256-bit) random output



Some notes on the generator

- In case of a true random number generator that generates 128-bit blocks, we expect a repeating value after around 2^{64} blocks had been generated
- In case of Fortuna, the output starts repeating after 2^{128} blocks
- For this reason, the number of blocks generated under a single key must be limited, to make it more difficult to detect the above deviation from a real random source
- Maximum number of blocks that can be requested from the generator in a single query is 2^{16} (this means 2^{20} bytes)
  - more random blocks can be generated by multiple queries to the generator
  - the generator's key K is changed between those queries (see previous slide)
- For a true random generator, the probability of a block collision in 2^{16} blocks is ~2^{-97}
- absence of collisions in case of Fortuna would not be detected until ~2^{97} requests had been made to the generator!

Re-seed mechanism

- The re-seed operation updates the state with an arbitrary input
- Re-seed algorithm:
  - input: any byte string S (i.e., the new seed)

    $K = H(K|S)$

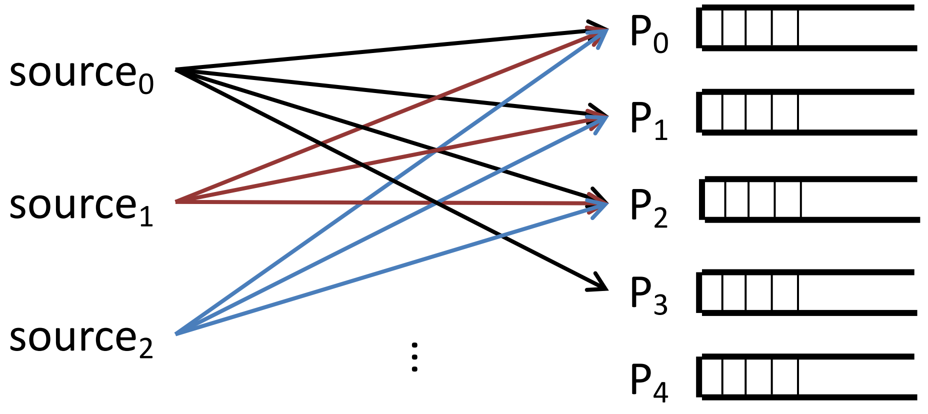
    $C = C+1$

  where H(.) is a hash function that produces 256-bit outputs (e.g., members of the SHA-2 or SHA-3 family)

- If either K or S is unguessable to the attacker, then the new key will also be unguessable!

Accumulator

- There are 32 pools P_0, P_1, ... P_{31}, in which we collect input samples from as many sources as possible
  - we append a new sample to the already collected samples in the pool
  - implementations do not need to store the unbounded string of samples, but compute the hash of the string incrementally
- Each source distributes its samples over the pools in a cyclic manner
  - thus, entropy from each source is distributed evenly over the pools



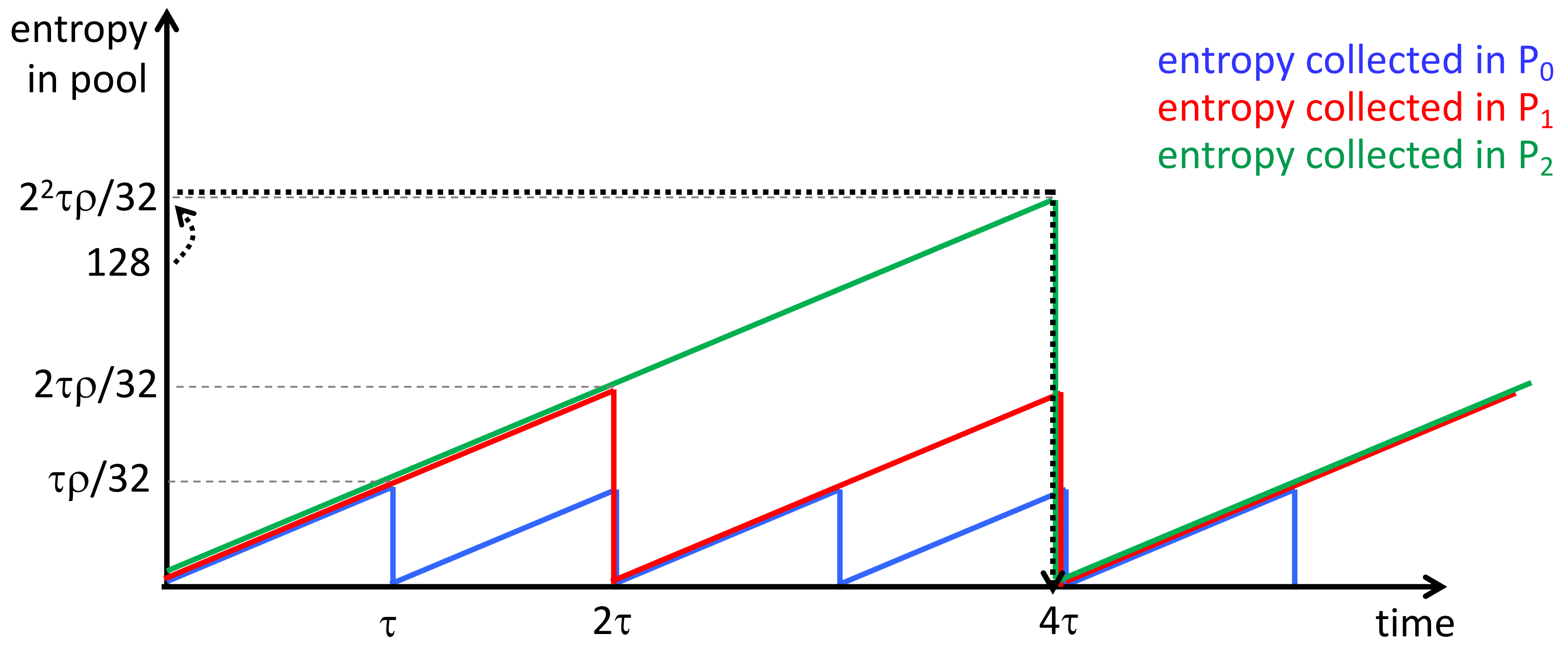
Re-seed control

- We re-seed the PRNG periodically, e.g., in every 100 ms
- Re-seed operations are numbered sequentially as 1, 2, 3, ...
- During re-seed r, pool P_i is used if 2^i is a divisor of r, hence:
  - P_0 is used in every re-seed
  - P_1 is used in every second re-seed
  - P_2 is used in every 4th re-seed
  - ...
- Using a pool in a re-seed means that its content is included in the new seed
- After a pool is used in a re-seed, it is cleared

Analysis of the re-seed control

- Let's assume that entropy flows in the system at fix rate ρ [bit/sec]
- Time between re-seed events is τ [sec]
- Entropy available in pool P_i when it is used in a reseed is $2^i\tau\rho/32$
- We may assume that the generator is put in an unguessable state if the re-seed involves a pool that contains at least 128 bits of entropy
- How long should we wait for being in an unguessable state?

Analysis of the re-seed control



smallest i for which $2^i\tau\rho/32 > 128$ determines when we will first re-seed from a pool that has more than 128 bits of entropy → $2^i\tau$

Analysis of the re-seed control

Numerical examples: (τ = 100ms)

$2^i\tau\rho/32$

i	0	1	2	3	4	5	6	7
ρ								
5120	16	32	64	128	256	512	1024	2048
1024	3,2	6,4	12,8	25,6	51,2	102,4	204,8	409,6
512	1,6	3,2	6,4	12,8	25,6	51,2	102,4	204,8

$2^i\tau$ = 800 ms

$2^i\tau$ = 6,4 s

$2^i\tau$ = 12,8 s

Attacker models for PRNGs

- Potential attacker goals:
  - predict future outputs
  - compute previous outputs (not yet observed)
  - recover the PRNGs internal state

- Possible attacker capabilities:
  - attacker can observe only some outputs
  - attacker can observe or manipulate collected input samples
  - attacker can occasionally compromise the internal state

Key derivation

Key derivation functions (KDFs)

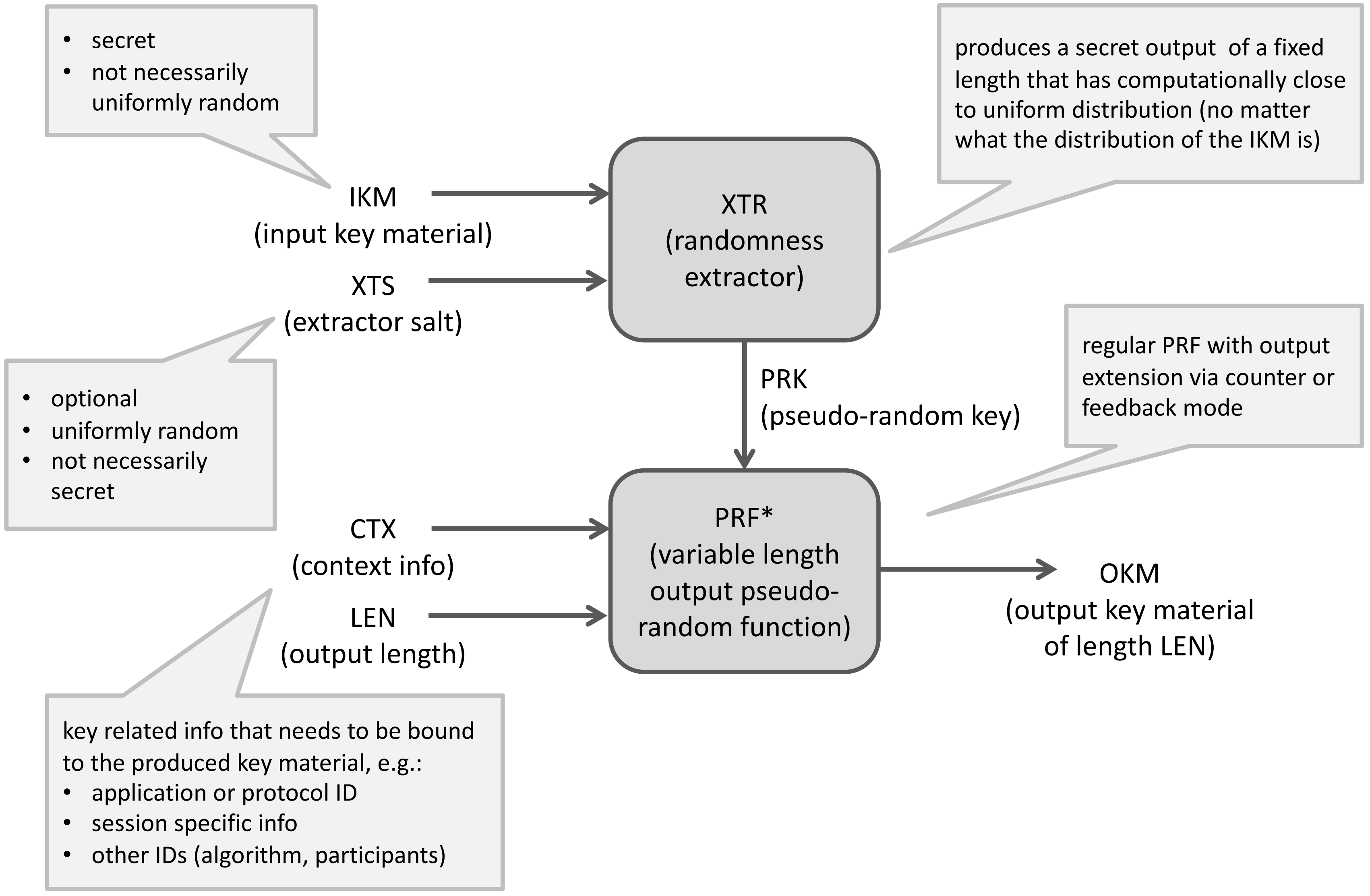
- Create one or more cryptographically strong secret keys from some initial key material

- The initial key material should already contain a good amount of randomness, but it may not be distributed uniformly, or an attacker may have some partial knowledge of it

  examples:
  - output of an imperfect physical random number generator
  - output of a statistical sampler (e.g., sampling system events or user keystrokes)
  - output of system PRNGs that use renewable sources of randomness
  - DH value (g^{xy}) computed as the result of DH key agreement

- KDFs can be used to create different keys needed by a communication session (e.g., encryption key, MAC key, keys for different directions) from some shared secret established by a key exchange protocol

Extract-then-expand approach

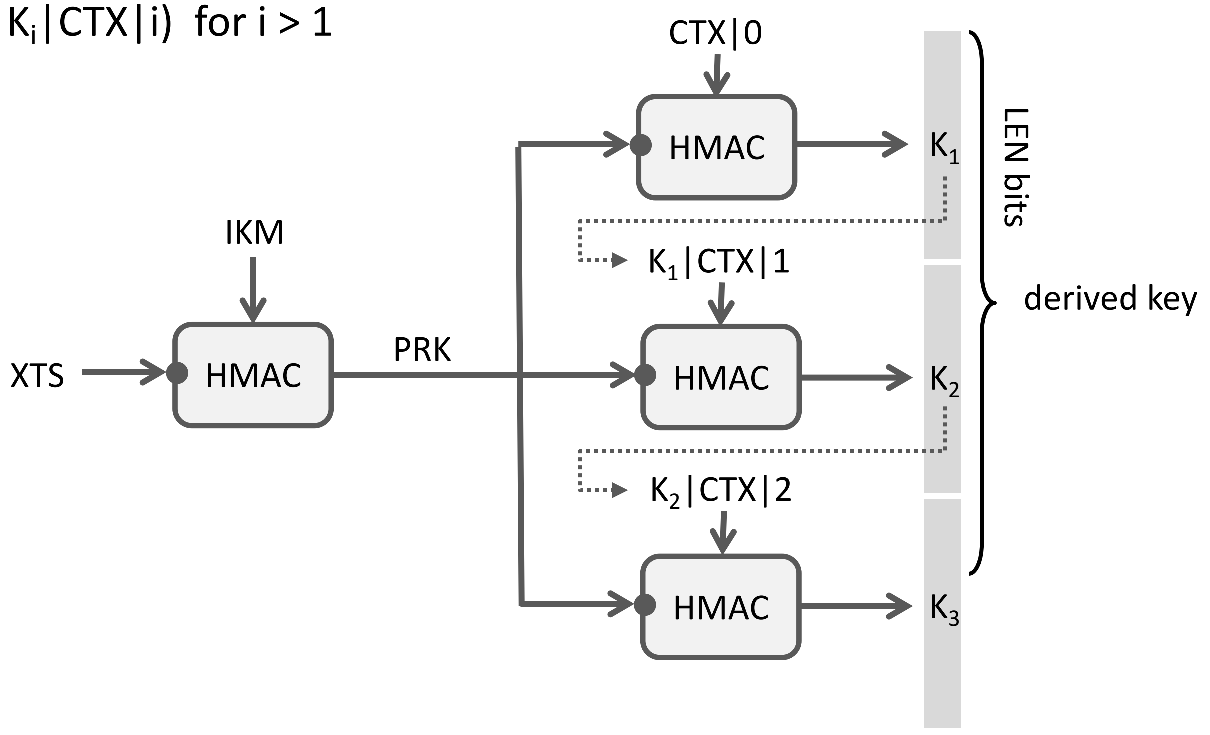


• secret
• not necessarily uniformly random
IKM (input key material)
XTS (extractor salt)
• optional
• uniformly random
• not necessarily secret
XTR (randomness extractor)
produces a secret output of a fixed length that has computationally close to uniform distribution (no matter what the distribution of the IKM is)
PRK (pseudo-random key)
regular PRF with output extension via counter or feedback mode
CTX (context info)
LEN (output length)
PRF* (variable length output pseudo-random function)
OKM (output key material of length LEN)
key related info that needs to be bound to the produced key material, e.g.:
• application or protocol ID
• session specific info
• other IDs (algorithm, participants)

HKDF

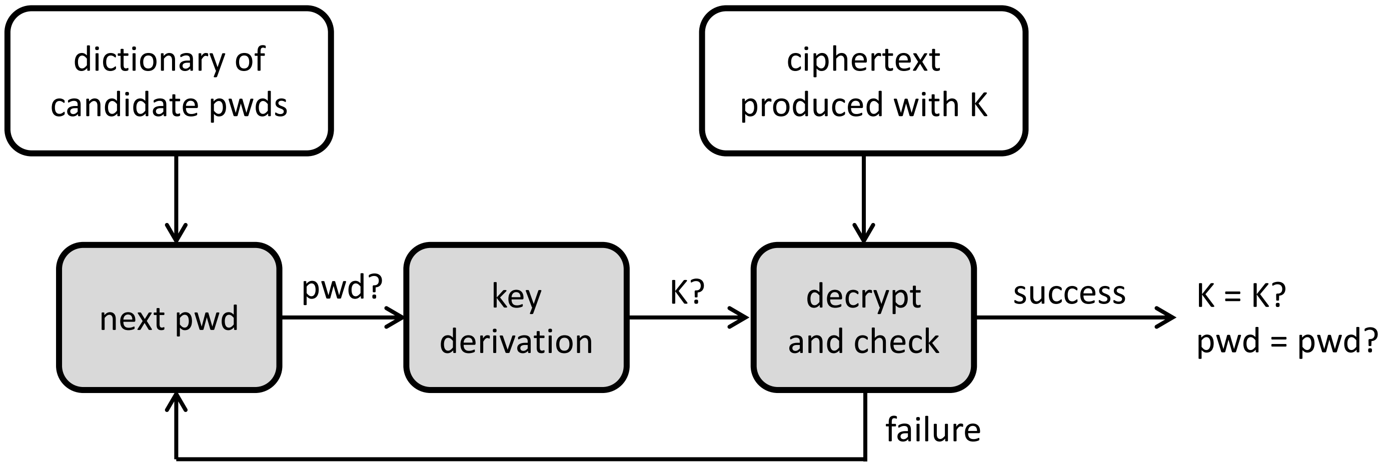
- HKDF = HMAC-based KDF
  - both XTR and PRF* are implemented with HMAC
  - HKDF(IKM, XTS, CTX, LEN) = K_1 | K_2 | ...

    where
    - » PRK = HMAC(XTS, IKM)
    - » K_1 = HMAC(PRK, CTX|0)
    - » K_{i+1} = HMAC(PRK, K_i|CTX|i) for i > 1



Password based key derivation

- We use passwords in many applications
- Can we use passwords as shared secret keys for cryptographic purposes? Or, **can we derive** such **keys from passwords**?
- A problem to cope with is that passwords are often chosen from a relatively small effective space (compared to crypto keys)
- Therefore, special care is required when using passwords to derive cryptographic keys in order to defend against **dictionary attacks**



Design considerations

- Stretching
  - Deliberately slow down key derivation, and hence, increase the cost of password guessing attacks, by using an iterative and inherently sequential key derivation process that repeats some underlying function many times
  - A modest number of iterations (e.g., 10000) is not likely to be a burden for legitimate parties when computing a key, but it will be a significant burden for attackers (e.g., making attacks 10000 times slower)

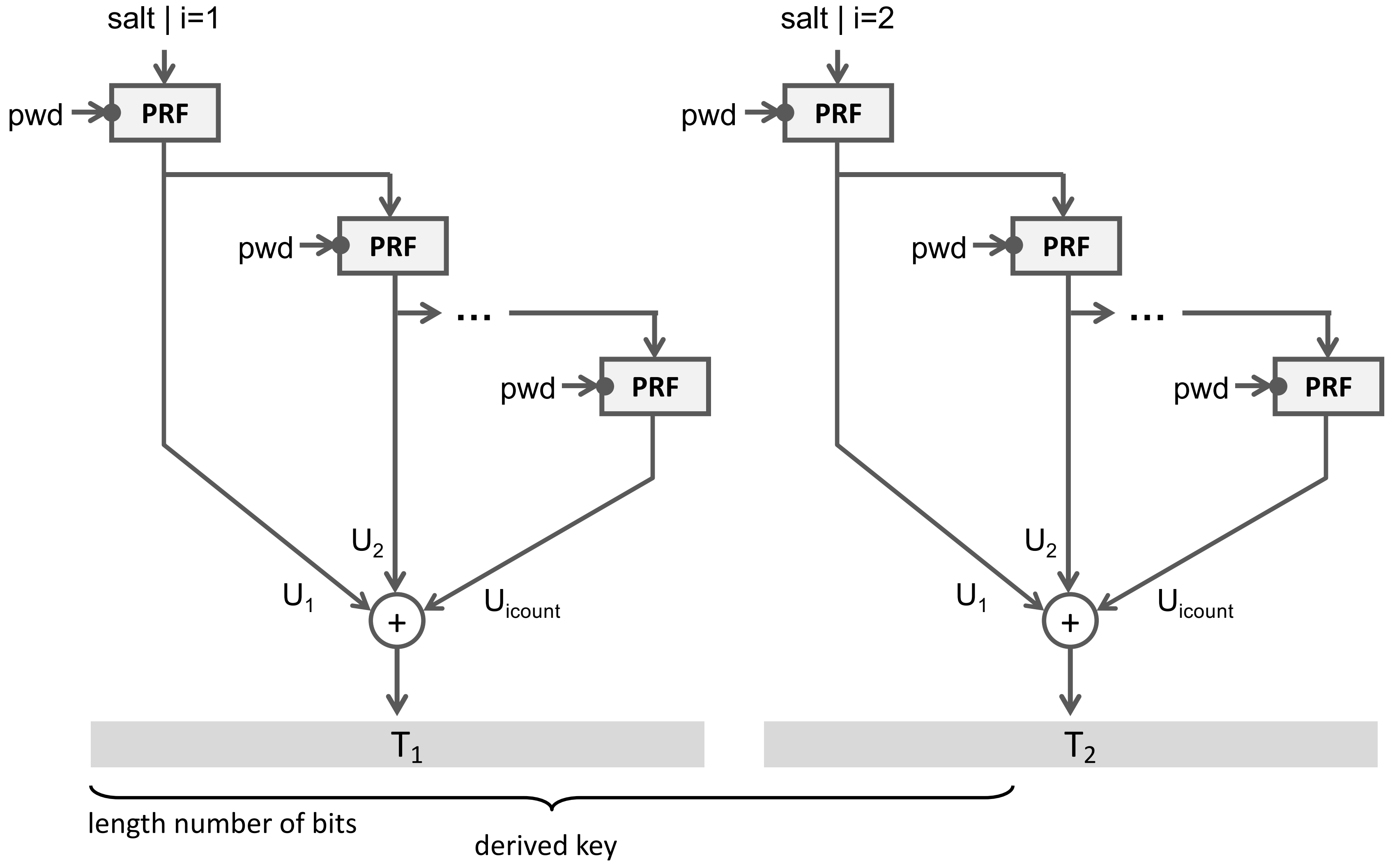
Design considerations

- Salting
  - A salt is a random (but not necessarily secret) value added to the computation of the key along with the password
  - Salting ensures that there are many possible keys derivable from a single password
    - if the salt is n bits long, the chance of “collision” between keys does not become significant until about $2^{n/2}$ keys have been produced (Birthday Paradox)
    - if n is large enough (e.g., 256), then it is unlikely that the same key will be derived twice from the same password
  - Salting also makes pre-computation attacks infeasible
    - an attacker who tries to pre-compute and store keys belonging to a dictionary of candidate passwords, now has to perform the pre-computation for all possible salt values and store all possible keys for each candidate password
    - if the salt is long enough, then pre-computation and storage of possible keys can easily become infeasible

PBKDF2

- PBKDF = Password Based Key Derivation Function
- derived key = PBKDF2(PRF, passwd, salt, icount, length)
  - PRF – a pseudorandom function with two inputs (e.g., HMAC-SHA256)
  - passwd – the password from which the key is being derived
  - salt – a random value of sufficient length (e.g., 256 bits)
  - icount – desired number of iterations (e.g., 10000 or more)
  - length – desired length of the derived key (e.g., 128 bits)
- computations:
  - PBKDF2(PRF, passwd, salt, icount, length) = $T_1 \mid T_2 \mid \ldots$
    where T_i = F(passwd, salt, icount, i)
  - F(passwd, salt, icount, i) = $U_1 \oplus U_2 \oplus \ \ldots \oplus U_{icount}$
    where U_1 = PRF(passwd, salt|i) and U_k = PRF(passwd, U_{k-1}) for k = 2, 3, ...

PBKDF2 – illustrated

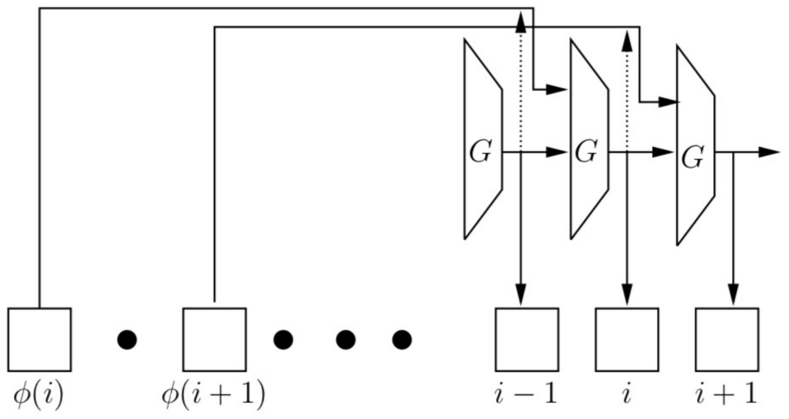


scrypt

- PBKDF2 has relatively low resource demands
  - does not require elaborate hardware or lot of memory to compute
  - can be easily and cheaply implemented in hardware (ASIC or FPGA)
  - large-scale parallel attack is possible → reduced amount of time needed to complete a brute-force attack
- scrypt is designed to hinder such attempts by requiring a large amount of memory for the key derivation procedure
  - a large vector of pseudorandom bits is generated as part of the algorithm
  - once this vector is generated, the elements of it are accessed in a pseudorandom order, and combined to produce the derived key
  - for fast computation, an implementation would need to keep the entire vector in random-access memory so that it can be accessed as needed
  - or the elements of the vector can be generated on-the-fly, but that would considerably slow down the computation (time-memory trade-off)
- see http://en.wikipedia.org/wiki/Scrypt for details and references...

Argon2*

- Winner of the Password Hashing Competition in July 2015
- Designed to reduce the possibility of time-memory trade-off attacks and to resist GPU cracking attacks
- Uses large amounts of memory and iterative computation
- In each iteration, it reads from a random memory location that has been written in previous iterations and writes in a memory location that will be read later



* more info, spec, and imlementation: https://github.com/P-H-C/phc-winner-argon2

PAKE

Password authenticated key exchange

- similar problem as before:
  - how to set up a cryptographic key between two parties with the help of a password that they already share?
- different solution
  - provides stronger security guarantees than PBKDF2
    - in case of PBKDF2, the work factor of the attacker is determined by the iteration count
    - but how many iterations can be considered sufficiently many?
    - if you underestimate this number, the attacker wins
    - if you overestimate it, then you unnecessarily increase the work factor of the legitimate users (computing keys will be slower for them as well)
    - in case of password authenticated key exchange protocols, the work factor of the attacker will always be high and independent of the work factor of legitimate users
  - but requires interaction of parties (running a protocol), and careful implementation

Encrypted Key Exchange (EKE)

- Alice generates a public key / private key pair K^+ and K^-, and encrypts K^+ with the hash of the password:

$$A \rightarrow B: \; A, \mathrm{Enc}_{H(pwd)}(K^+)$$

- Bob uses the hash of the password to obtain K^+, then generates a symmetric key K, and encrypts it with K^+ in the public key cryptosystem $\mathrm{PubEnc}_{...}(...)$; the result may be further encrypted with the hash of the password:

$$B \rightarrow A: \; \mathrm{Enc}_{H(pwd)}(\mathrm{PubEnc}_{K+}(K))$$

- Alice uses the hash of the password and K^- to obtain K from $\mathrm{Enc}_{H(pwd)}(\mathrm{PubEnc}_{K+}(K))$; then she can use K to send messages to Bob:

$$A \rightarrow B: \; \mathrm{Enc}_K(\text{“Last login at 16:34, Monday”})$$

Why is this good?

- For a candidate password pwd?, the attacker can compute a candidate public key K^+? as $Dec_{H(pwd?)}(Enc_{H(pwd)}(K^+))$
- But K^+? cannot really be tested on $PubEnc_{K+}(K)$
  - the attacker needs to find a key K? such that
    - » $PubEnc_{K+?}(K?) = Dec_{H(pwd?)}(Enc_{H(pwd)}(PubEnc_{K+}(K)))$, or
    - » $Dec_{K?}(Enc_K($"Last login …"$))$ makes sense
  - **both would require an exhaustive search over the key space from which K is chosen** (or breaking the symmetric or the asymmetric cipher)

→ the relatively small space of passwords is thus multiplied by the large key space from which K is chosen (privacy amplification effect)

Implementation of EKE is non-trivial

- Let’s assume we want to use RSA as PubEnc()
  - public key is (e, n), where n = pq
- Problem with encrypting (e, n) in the first message
  - the attacker can compute (e?, n?) = $\mathrm{Dec}_{H(pwd?)}(\mathrm{Enc}_{H(pwd)}((e, n))$, and check if n? has a small prime factor
  - if pwd? is not the correct password, then n? is random and it has a small prime factor with high probability
  - thus, the attacker can throw away wrong password candidates easily
- If Enc() is a block cipher used in CBC mode, then using non-random padding is vulnerable
  - for any candidate password pwd?, the attacker can try to decrypt $\mathrm{Enc}_{H(pwd)}(K^{+})$ with H(pwd?) and check if the padding obtained is valid

Secure Remote Password (SRP) protocol

- SRP is an asymmetric PAKE protocol
  - Bob does not need to know the password of Alice (or the hash of it)
  - Bob only stores a verification value that is a one-way function of the pwd
  - Alice proves her knowledge of the password in a way that does not reveal anything useful about the password apart from the fact that Alice knows it

- Notation and preparations:
  - q and $p = 2q+1$ are primes
  - g is a generator of Z_p^*
  - $H()$ is a hash function (e.g., SHA-256)
  - $k = H(p|g)$
  - Alice's user name and password are ID_{Alice} and PW_{Alice}
  - $x = H(s \mid H(ID_{\text{Alice}}|":"|PW_{\text{Alice}}))$ where s is a salt
  - $v = g^x$ is the password verifier value stored together with s by Bob
  - g^x means $g^x \bmod p$, of course

Secure Remote Password (SRP) protocol

1. Alice generates random value a and sends $A = g^a$ to Bob
2. Bob generates random value b and send s and $B = kv + g^b$ to Alice
3. Both Alice and Bob compute $u = H(A|B)$
4. Alice computes

   $x = H(s \mid H(ID_{\mathrm{Alice}}|\text{":"}|PW_{\mathrm{Alice}}))$

   $S_{\mathrm{Alice}} = (B - kg^x)^{(a + ux)} = (kv + g^b - kg^x)^{(a + ux)} = (kg^x - kg^x + g^b)^{(\mathrm{a + ux})} = g^{(a + ux)b}$

   $K_{\mathrm{Alice}} = H(S_{\mathrm{Alice}})$
5. Bob computes

   $S_{\mathrm{Bob}} = (Av^u)^b = (g^a v^u)^b = (g^a(g^x)^u)^b = (g^{\mathrm{a + ux}})^b = g^{(\mathrm{a + ux})b}$

   $K_{\mathrm{Bob}} = H(S_{\mathrm{Bob}}) = K_{\mathrm{Alice}}$
6. Alice computes and sends to Bob

   $M_1 = H(\, H(p) \oplus H(g) \mid H(ID_{\mathrm{Alice}}) \mid s \mid A \mid B \mid K_{\mathrm{Alice}}\,)$
7. Bob computes and sends to Alice

   $M_2 = H(A \mid M_1 \mid K_{\mathrm{Bob}})$
8. Alice and Bob verifies M_1 and M_2, respectively.

Summary

- Good quality (unpredictable) random numbers are essential in cryptographic applications
- Harvesting true random bits can be slow --» random bits may not be available when we need them (e.g., when generating keys)
- Pseudo-random number generators (PRNGs) generate pseudo-random bits in large quantity and fast enough, and if designed properly, they maintain practical unpredictability
- PRNGs do need true random inputs, but they use them only for seeding the generation process
- Attacker model for PRNGs
  - <u>goals:</u> predict unobserved outputs, determine internal state, extend state compromise
  - <u>capabilities:</u> observe some outputs < observe some inputs < manipulate some inputs < occasionally compromise the internal state

Summary

- We often need to derive keys from already shared strong or weak secrets
- Key derivation from a strong secret can be based on the extract-and-expand approach
  - extract phase derives a pseudo-random key from the secret
  - expand phase expands the PRK to as many bits as needed
  - an implenetation is the HKDF function
- Key derivation from a weak secret may be vulnerable to off-line guessing and dictionary attacks
  - stretching deliberately slows down the derivation process
  - salting prevents pre-computation attacks
  - PBKDF2, scrypt, and Argon2 are practical implementations
- Password authenticated key exchange (e.g., EKE, SRP) solves a similar problem in a different way with some security advantages

Control questions

- Why do we need random numbers in cryptographic applications?
- What is the difference between a true random number generator and a pseudo-random number generator (PRNG)?
- What practical problems do PRNGs solve?
- What is the general structure of a PRNG?
- What do we assume about the attacker in case of PRNGs? (goals and capabilities)
- How does the Fortuna PRNG work?
  - internal state (what makes up the internal state?)
  - generation algorithm (how outputs are generated?)
  - entropy accumulator (how true random samples are collected?)
  - re-seed control (when re-seeds happen?)
  - re-seed algorithm (how re-seeding works?)

Control questions

- Why do we need to derive keys from an already random shared secret?
- How does the extract-then-expand key derivation approach work?
- How does HKDF implements the extract-then-expand approach?
- What is the basic problem we need to consider when deriving cryptographic keys from passwords?
- In case of password based key derivation, what do we mean by "stretching" and "salting"? Why is it advantageous to apply them?
- How does PBKDF2 work?
- What are the advantages and disadvantages of an interactive password authenticated key exchange protocol with respect to an off-line password based key derivation function?
- How does Encrypted Key Exchange (EKE) work? How does it prevent dictionary attacks on the password?
- What is the key advantage of the Secure Remote Password (SRP) protocol compared to EKE?