

User Authentication and Cryptographic Primitives

Brad Karp

UCL Computer Science



CS GZ03 / M030

17th November, 2010

Outline

- **Authenticating users**
 - Local users: hashed passwords
 - Remote users: s/key
 - Unexpected covert channel: the Tenex password-guessing attack
- Symmetric-key-cryptography
- Public-key cryptography usage model
- RSA algorithm for public-key cryptography
 - Number theory background
 - Algorithm definition

Dictionary Attack on Hashed Password Databases

- Suppose hacker obtains copy of password file (until recently, world-readable on UNIX)
- Compute $H(x)$ for 50K common words
- String compare resulting hashed words against passwords in file
- **Learn all users' passwords that are common English words after only 50K computations of $H(x)$!**
- **Same hashed dictionary works on all password files in world!**

Salted Password Hashes

- Generate a random string of bytes, r
- For user password x , store $[H(r,x), r]$ in password file
- Result: same password produces **different result on every machine**
 - So must see password file before can hash dictionary
 - ...and single hashed dictionary won't work for multiple hosts
- Modern UNIX: password hashes **salted**; hashed password database **readable only by root**

Salted Password Hashes

- Generate a random string of bytes, r

Dictionary attack still possible after attacker sees password file!

Users should pick passwords that aren't close to dictionary words.

- So must see password file before can hash dictionary
- ...and single hashed dictionary won't work for multiple hosts
- Modern UNIX: password hashes **salted**; hashed password database **readable only by root**

Tenex Password Attack: An Information Leak

- Tenex OS stored directory passwords in **cleartext**
- OS supported system call:
 - `pw_validate(directory, pw)`
- Implementation simply compared pw to stored password in directory, char by char
- Clever attack:
 - Make pw span two VM pages, put 1st char of guess in first page, rest of guess in second page
 - See whether get a page fault—if not, try next value for 1st char, &c.; if so, first char correct!
 - Now position 2nd char of guess at end of 1st page, &c.
 - **Result: guess password in time linear in length!**

Tenex Password Attack: An Information Leak

- Tenex OS stored directory passwords in **cleartext**

Lessons:

Don't store passwords in cleartext.

Information leaks are real, and can be extremely difficult to find and eliminate.

- Make pw span two VM pages, put 1st char of guess in first page, rest of guess in second page
- See whether get a page fault—if not, try next value for 1st char, &c.; if so, first char correct!
- Now position 2nd char of guess at end of 1st page, &c.
- **Result: guess password in time linear in length!**

Remote User Authentication

- Consider the case where Alice wants to log in **remotely**, across LAN or WAN from server
- Suppose network links can be **eavesdropped** by adversary, Eve
- Want scheme immune to **replay**: if Eve overhears messages, shouldn't be able to log in as Alice by repeating them to server
- Clear non-solutions:
 - Alice logs in by sending {alice, password}
 - Alice logs in by sending {alice, H(password)}

Remote User Authentication (2)

- Desirable properties:
 - Message from Alice must change unpredictably at each login
 - Message from Alice must be verifiable at server as matching secret value known only to Alice
- Can we achieve these properties using **only** a cryptographic hash function?

Remote User Authentication: s/key

- Denote by $H^n(x)$ n successive applications of cryptographic hash function $H()$ to x
 - i.e., $H^3(x) = H(H(H(x)))$
- Store in server's user database:
`alice:99:H99(password)`
- At first login, Alice sends:
`{alice, H98(password)}`
- Server then updates its database to contain:
`alice:98:H98(password)`
- At next login, Alice sends:
`{alice, H97(password)}`
 - and so on...

Properties of s/key

- Just as with any hashed password database, Alice must store her secret on the server securely (best if physically at server's console)
- Alice must choose total number of logins at time of storing secret
- When logins all "used", must store new secret on server securely again

Secrecy through Symmetric Encryption

- Two functions: $E()$ encrypts, $D()$ decrypts
- Parties share secret key K
- For message M :
 - $E(K, M) \rightarrow C$
 - $D(K, C) \rightarrow M$
- M is plaintext; C is ciphertext
- Goal: attacker cannot derive M from C without K

Idealized Symmetric Encryption: One-Time Pad

- Secretly share a **truly random bit string** P at sender and receiver
- Define \oplus as bit-wise XOR
- $C = E(M) = M \oplus P$
- $M = D(C) = C \oplus P$
- Use bits of P only once; **never use them again!**

Stream Ciphers: Pseudorandom Pads

- Generate pseudorandom bit sequence (stream) at sender and receiver from short key
- Encrypt and decrypt by XOR'ing message with sequence, as with one-time pad
- Most widely used stream cipher: RC4
- Again, **never, ever re-use bits from pseudorandom sequence!**
- What's wrong with reusing the stream?
 - Alice → Server: $c_1 = E(s, \text{"Visa card number"})$
 - Server → Alice: $c_2 = E(s, \text{"Transaction confirmed"})$
 - Suppose Eve hears both messages
 - Eve can compute:
 $m = c_1 \oplus c_2 \oplus \text{"Transaction confirmed"}$

Symmetric Encryption: Block Ciphers

- Divide plaintext into **fixed-size blocks** (typically 64 or 128 bits)
- Block cipher maps each plaintext block to **same-length ciphertext block**
- Best today to use **AES** (others include Blowfish, DES, ...)
- Of course, message of arbitrary length; **how to encrypt message of more than one block?**

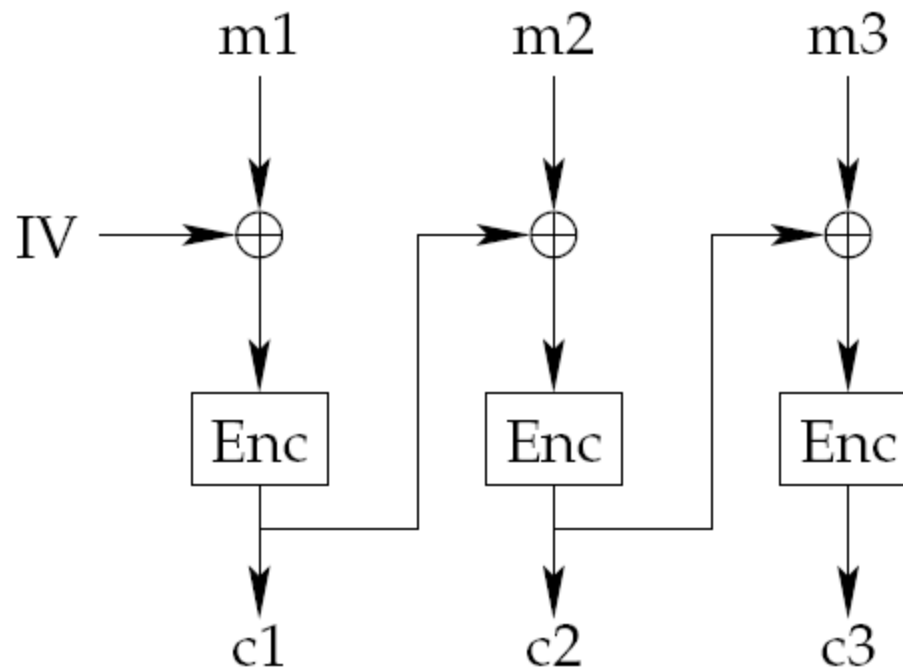
Using Block Ciphers: ECB Mode

- Electronic Code Book method
- Divide message M into blocks of cipher's block size
- Simply encrypt each block individually using the cipher
- Send each encrypted block to receiver
- Presume cipher provides secrecy, so attacker cannot decrypt any block
- **Does ECB mode provide secrecy?**

Avoid ECB Mode!

- ECB mode **does not provide robust secrecy!**
- What if there are repeated blocks in the plaintext? **Repeated as-is in ciphertext!**
- What if sending sparse file, with long runs of zeroes? **Non-zero regions obvious!**
- WW II U-Boat example (Bob Morris):
 - Each day at same time, when no news, send encrypted message: “Nichts zu melden.”
 - When there’s news, send the news at that time.
 - **Obvious when there’s news**
 - **Many, many ciphertexts of same known plaintext made available to adversary for cryptanalysis—a worry even if encryptions of same plaintext produce different ciphertexts!**

Using Block Ciphers: CBC Mode



- Better plan: make encryptions of successive blocks **depend on one another, and initialization vector known to receiver**

Integrity with Symmetric Crypto: Message Authentication Codes

- How does receiver know if message modified en route?
- Message Authentication Code:
 - Sender and receiver share secret key K
 - On message M , $v = \text{MAC}(K, M)$
 - Attacker cannot produce valid $\{M, v\}$ without K
- Append MAC to message for **tamper-resistance**:
 - Sender sends $\{M, \text{MAC}(K, M)\}$
 - M could be ciphertext, $M = E(K', m)$
 - Receiver of $\{M, v\}$ can verify that $v = \text{MAC}(K, M)$
- Beware replay attacks—replay of prior $\{M, v\}$ by Eve!

HMAC: A MAC Based on Cryptographic Hash Functions

- $\text{HMAC}(K, M) = H(K \oplus \text{opad} . H(K \oplus \text{ipad} . M))$
- where:
 - . denotes string concatenation
 - opad = 64 repetitions of 0x36
 - ipad = 64 repetitions of 0x5c
 - $H()$ is a cryptographic hash function, like SHA-256
- Fixed-size output, even for long messages

Public-Key Encryption: Interface

- Two keys:
 - Public key: K , published for all to see
 - Private (or secret) key: K^{-1} , kept secret
- Encryption: $E(K, M) \rightarrow \{M\}_K$
- Decryption: $D(K^{-1}, \{M\}_K) \rightarrow M$
- Provides **secrecy**, like symmetric encryption:
 - Can't derive M from $\{M\}_K$ without knowing K^{-1}
- Same public key used by all to encrypt **all messages to same recipient**
 - Can't derive K^{-1} from K

Number Theory Background: Modular Arithmetic Primer (1)

- Recall the “mod” operator: returns **remainder** left after dividing one integer by another, the **modulus**
 - e.g., $15 \bmod 6 = 3$
- That is:
 - $a \bmod n = r$
 - which just means
 - $a = kn + r$ for some integers k and r
- Note that $0 \leq r < n$

Modular Arithmetic Primer (2)

- In **modular arithmetic**, constrain range of integers to be only the **residues** $[0, n-1]$, for modulus n
 - e.g., $(12 + 13) \bmod 24 = 1$
 - We may also write $12 + 13 \equiv 1 \pmod{24}$
- Modular arithmetic retains familiar properties: **commutative, associative, distributive**
- Same results whether mod taken at each arithmetic operation, or only at end, e.g.:
 - $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
 - $(ab) \bmod n = (a \bmod n)(b \bmod n) \bmod n$

Modular Arithmetic: Advantages

- Limits precision required: working mod n , where n is k bits long, any single arithmetic operation yields **at most $2k$ bits**
 - ...so results of even seemingly expensive ops, like exponentiation (a^x) fit in **same number of bits as original operand(s)**
 - Lower precision means **faster arithmetic**
- Some operations in modular arithmetic are **computationally very difficult**:
 - e.g., computing **discrete logarithms**:
find integer x s.t. $a^x \equiv b \pmod{n}$

Modular Arithmetic: Advantages

- Limits precision required: working mod n , where n is k bits long, any single arithmetic operation yields **at most $2k$ bits**

Cryptography leverages “difficult” operations; want reversing encryption without key to be computationally intractable!

- Some operations in modular arithmetic are **computationally very difficult**:
 - e.g., computing **discrete logarithms**:
find integer x s.t. $a^x \equiv b \pmod{n}$

Modular Arithmetic: Inverses (1)

- In real arithmetic, every integer has a multiplicative inverse—its reciprocal—and their product is 1
 - e.g., $7x = 1 \rightarrow x = (1/7)$
- What does an inverse in modular arithmetic (say, mod 11) look like?
 - $7x \equiv 1 \pmod{11}$
 - that is, $7x = 11k + 1$ for some x and k
 - so $x = 8$ (where $k = 5$)

Aside: Prime Numbers

- Recall: **prime number** is integer > 1 that is evenly divisible only by 1 and itself
- Two integers a and b are **relatively prime** if they share no common factors but 1; i.e., if $\gcd(a, b) = 1$
- There are infinitely many primes
- Large primes (512 bits and longer) figure prominently in public-key cryptography

Modular Arithmetic: Inverses (2)

- In general, finding modular inverse means finding x s.t. $a^{-1} \equiv x \pmod{n}$
- **Does modular inverse always exist?**
 - No! Consider $2^{-1} \equiv x \pmod{8}$
- In general, when a and n are relatively prime, modular inverse x exists and is unique
- When a and n not relatively prime, x doesn't exist
- When n prime, all of $[1 \dots n-1]$ relatively prime to n , and have an inverse in that range

Modular Arithmetic: Inverses (2)

In general, finding modular inverse means

Algorithm to find modular inverse: extended Euclidean Algorithm. Tractable; requires $O(\log n)$ divisions.

- In general, when a and n are relatively prime, modular inverse x exists and is unique
- When a and n not relatively prime, x doesn't exist
- When n prime, all of $[1 \dots n-1]$ relatively prime to n , and have an inverse in that range

Euler's Phi Function: Efficient Modular Inverses on Relative Primes

- $\phi(n)$ = number of integers $< n$ that are relatively prime to n
- If n prime, $\phi(n) = n-1$
- If $n=pq$, where p and q prime:
 $\phi(n) = (p-1)(q-1)$
- If a and n relatively prime, Euler's generalization of Fermat's little theorem:
 $a^{\phi(n)} \bmod n = 1$
- and thus, to find inverse x s.t. $x = a^{-1} \bmod n$:
 $x = a^{\phi(n)-1} \bmod n$

RSA Algorithm (1)

- [Rivest, Shamir, Adleman, 1978]
- Recall that public-key cryptosystems use two keys per user:
 - K , the **public key**, made available to all
 - K^{-1} , the **private key**, kept secret by user

RSA Algorithm (2)

- Choose two random, large primes, p and q , of equal length, and compute $n=pq$
- Randomly choose encryption key e , s.t. e and $(p-1)(q-1)$ are relatively prime
- Use extended Euclidean algorithm to compute d , s.t. $d = e^{-1} \text{ mod } ((p-1)(q-1))$
- Public key: $K = (e, n)$
- Private key: $K^{-1} = d$
- Discard p and q

RSA Algorithm (3)

- Encryption:
 - Divide message M into blocks m_i , each shorter than n
 - Compute ciphertext blocks c_i with:
$$c_i = m_i^e \bmod n$$
- Decryption
 - Recover plaintext blocks m_i with:
$$m_i = c_i^d \bmod n$$

Why Does RSA Decryption Recover Original Plaintext?

- Observe that $c_i^d = (m_i^e)^d = m_i^{ed}$
- Note that $ed \equiv 1 \pmod{(p-1)(q-1)}$
because e and d are inverses mod $(p-1)(q-1)$
- So:
 $ed \equiv 1 \pmod{(p-1)}$, and thus $ed = k(p-1)+1$
 $ed \equiv 1 \pmod{(q-1)}$, and thus $ed = h(q-1)+1$
- Consider case where m_i and p are relatively prime:
 $m_i^{(p-1)} \equiv 1 \pmod{p}$ by Euler's generalization of Fermat's little theorem
– so $m_i^{ed} = m_i^{k(p-1)+1} = m_i(m_i^{(p-1)})^k \equiv m_i \pmod{p}$
- And case where m_i a multiple of p :
 $m_i^{ed} = 0^{ed} = 0 \equiv m_i \pmod{p}$
- Thus in all cases, $m_i^{ed} \equiv m_i \pmod{p}$

Why Does RSA Decryption Recover Original Plaintext? (2)

- Similarly, $m_i^{ed} \equiv m_i \pmod{q}$
- Now:
$$m_i^{ed} - m_i \equiv 0 \pmod{p}$$
$$m_i^{ed} - m_i \equiv 0 \pmod{q}$$
- Because p, q both prime and distinct:
$$m_i^{ed} - m_i \equiv 0 \pmod{(pq)}$$
- So $c_i^d = m_i^{ed} \equiv m_i \pmod{n}$