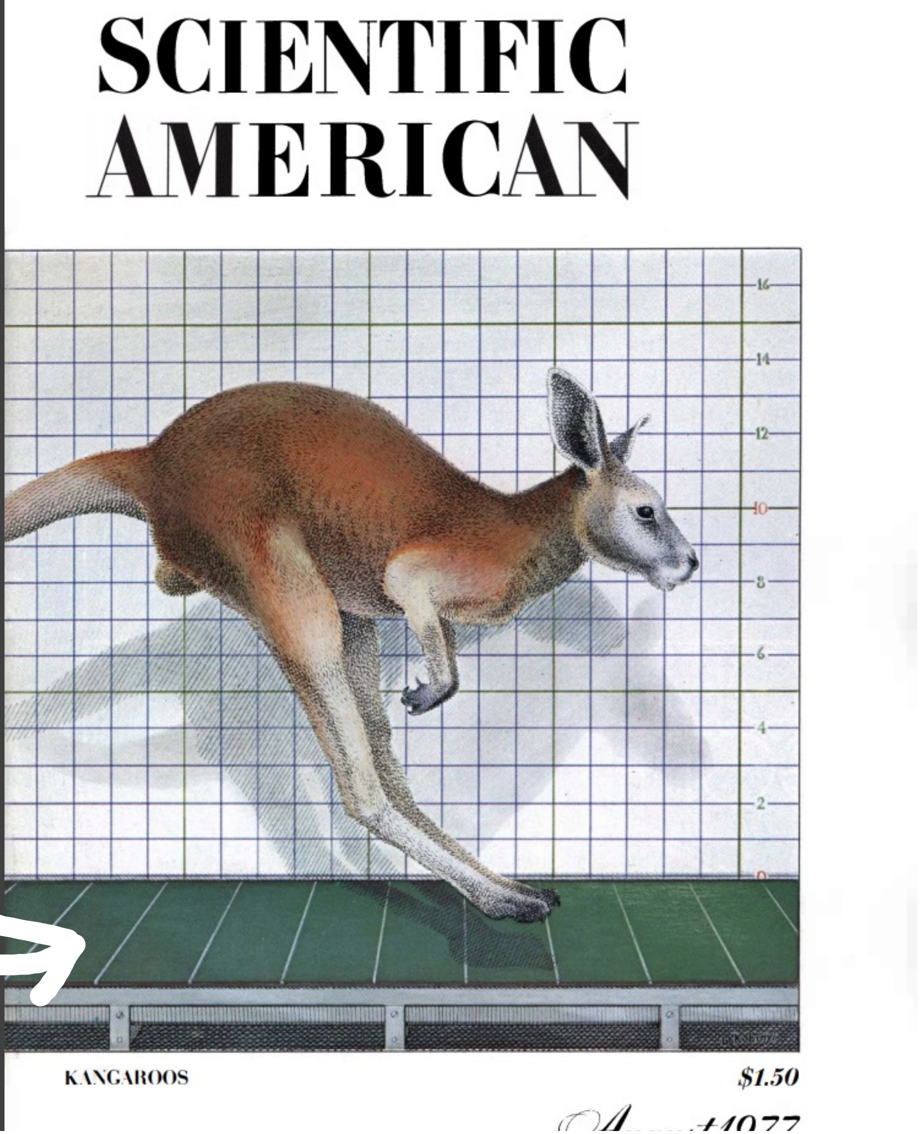
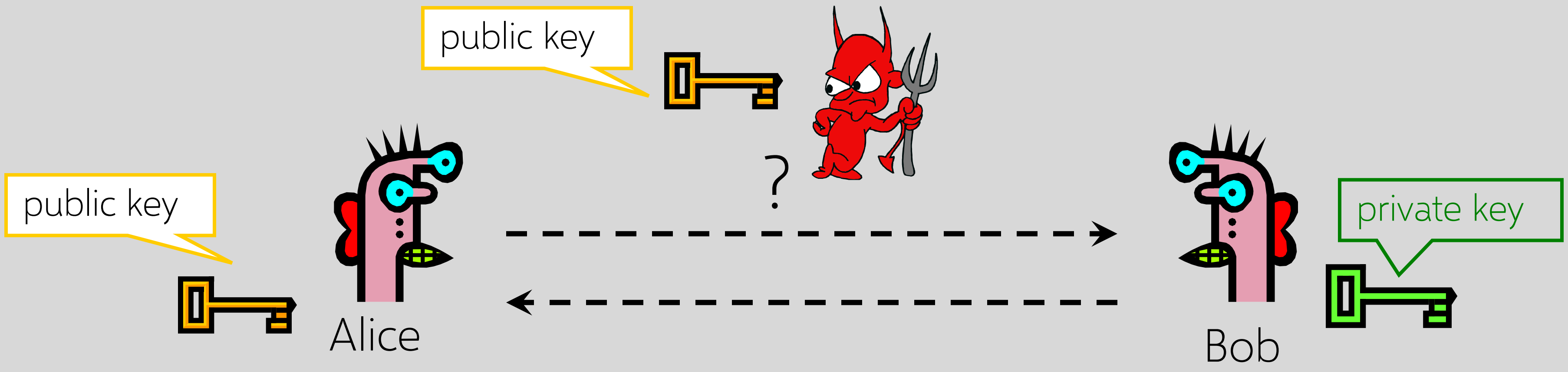
BASICS OF PUBLIC-KEY CRYPTOGRAPHY

VITALY SHMATIKOV

RSA was described for the first time in the August 1977 issue of "Scientific American"



Public-Key Cryptography



Anyone can encrypt a message
Only someone who knows the private key can decrypt
Secret keys are only stored in one place

Encryption for confidentiality

Digital signatures for authentication

Only someone who knows the private key can sign

Session key establishment

Exchange messages to create a secret session key
Then switch to symmetric cryptography (why?)

Public-Key Encryption

Key generation: computationally easy to generate a pair (public key PK, private key SK)

Encryption: given plaintext M and public key PK, easy to compute ciphertext $C=E_{PK}(M)$

Decryption: given ciphertext $C=E_{PK}(M)$ and private key SK, easy to compute plaintext M

- Infeasible to learn anything about M from C without SK
- "Trapdoor" function: Decrypt(SK,Encrypt(PK,M))=M

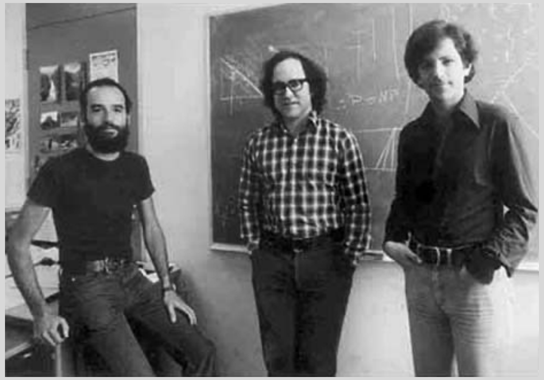
Some Number Theory Facts

- Euler totient function $\varphi(n)$ where $n \geq 1$ is the number of integers in the [1,n] interval that are relatively prime to n
  - Two numbers are relatively prime if their greatest common divisor (gcd) is 1
- Euler's theorem:

  if $a \in Z_n^*$, then $a^{\varphi(n)} \equiv 1 \bmod n$
- Special case: <u>Fermat's Little Theorem</u>

  if p is prime and $\gcd(a,p)=1$, then $a^{p-1} \equiv 1 \bmod p$

RSA Cryptosystem



Rivest, Shamir, Adleman

Key generation:

- Generate large primes p, q and compute n=pq
  - At least 2048 bits each... need primality testing!
  - Note that $\varphi(n)=(p-1)(q-1)$
- Choose small e, relatively prime to $\varphi(n)$
  - Typically, e=3 (may be vulnerable) or $e=2^{16}+1=65537$ (why?)
- Compute unique d such that $ed \equiv 1 \bmod \varphi(n)$
- Public key = (e,n); private key = d

Encryption of m: $c = m^e \bmod n$

Decryption of c: $c^d \bmod n = (m^e)^d \bmod n = m$

Why RSA Decryption Works

- $e \cdot d \equiv 1 \bmod \varphi(n)$
- Thus $e \cdot d = 1 + k \cdot \varphi(n) = 1 + k(p-1)(q-1)$ for some k
- If $\gcd(m,p)=1$, then by Fermat's Little Theorem, $m^{p-1} \equiv 1 \bmod p$
- Raise both sides to the power $k(q-1)$ and multiply by m, obtaining $m^{1+k(p-1)(q-1)} \equiv m \bmod p$
- Thus $m^{ed} \equiv m \bmod p$
- By the same argument, $m^{ed} \equiv m \bmod q$
- Since p and q are distinct primes and $p \cdot q = n$,

  $m^{ed} \equiv m \bmod n$

Why Is RSA Secure?

RSA problem: given c, $n=pq$, and e such that $\gcd(e,(p-1)(q-1))=1$, find m such that $m^e=c \bmod n$

- That is, recover m from ciphertext c and public key (n,e) by taking e^{th} root of c modulo n
- There is no known efficient algorithm for doing this

Factoring problem: given positive integer n, find primes $p_1, \ldots, p_k$ such that $n=p_1^{e1}p_2^{e2}\ldots p_k^{ek}$

If factoring is easy, then RSA problem is easy, but may be possible (believed unlikely) to break RSA without factoring n

Factoring Records

RSA-x is an RSA challenge modulus of size x bits

Algorithm	Year	Algorithm	Time
RSA-400	1993	Quadratic sieve	830 MIPS years
RSA-478	1994	Quadratic sieve	5000 MIPS years
RSA-515	1999	Number-field sieve	8000 MIPS years
RSA-768	2009	Number-field sieve	~2.5 years

Nowadays, minimal recommended size is 2048-bit modulus

Exponentiation in $\mathcal{O}(\log N)$, and so size impacts performance

“Textbook” RSA Is Bad Encryption

- Deterministic
  - Attacker can guess plaintext, compute ciphertext, and compare for equality
  - If messages are from a small set (for example, yes/no), can build a table of corresponding ciphertexts
- Can tamper with encrypted messages, no integrity protection
  - Take an encrypted auction bid c and submit $c(101/100)^e \bmod n$ instead
- Does not provide security against chosen-plaintext attacks

Integrity in RSA Encryption

Always use standard hashing and padding with RSA… better yet, use a good library implementation

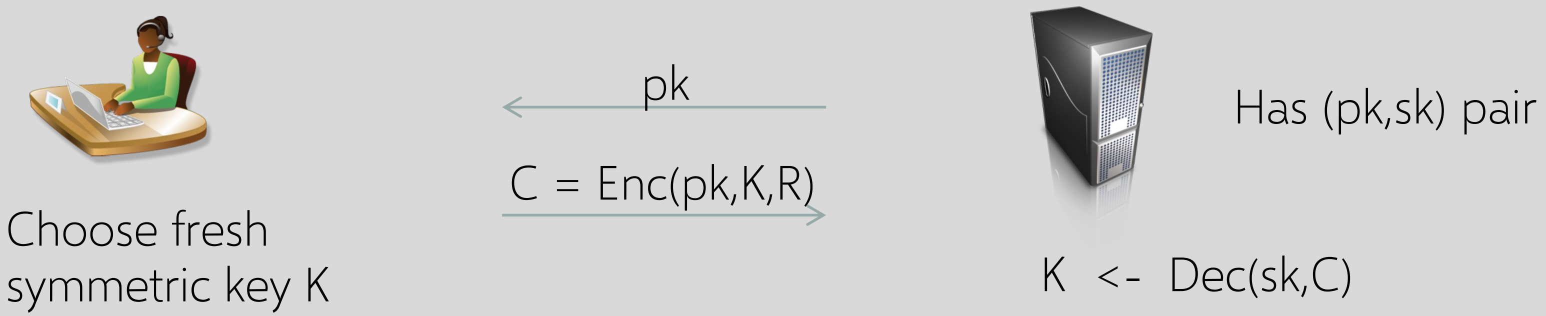
"Textbook" RSA does not provide integrity

- Given encryptions of m_1 and m_2, attacker can create encryption of $m_1 \cdot m_2$ because $(m_1^e) \cdot (m_2^e) \bmod n \equiv (m_1 \cdot m_2)^e \bmod n$
- Attacker can convert m into m^k without decrypting because $(m^e)^k \bmod n \equiv (m^k)^e \bmod n$

In practice, **OAEP** is used: instead of encrypting M, encrypt $M \oplus G(r)$; $r \oplus H(M \oplus G(r))$

- r is random and fresh, G and H are hash functions
- Resulting encryption is "plaintext-aware": infeasible to compute a valid encryption without knowing plaintext… assuming hash functions are "good" and the RSA problem is hard

Session Key Establishment



Server picks long-lived (pk,sk) pair; pk sent to client (how?)

Client encrypts a fresh session key K using pk and some fresh randomness R

Ciphertext C sent to server; server decrypts using sk

Forward Secrecy?

Choose fresh
symmetric key K

pk

C = Enc(pk,K,R)

Has (pk,sk) pair

K <- Dec(sk,C)

Record encrypted transcript

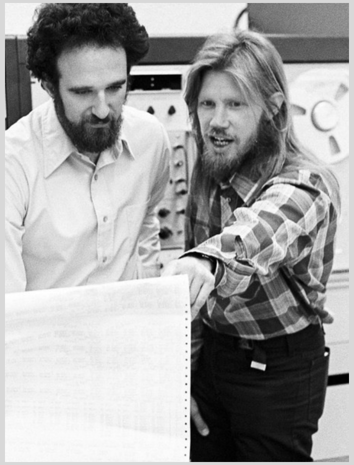
Sometime later… break in and steal sk

Can adversary recover K? Yes!

We want a key exchange protocol that provides forward secrecy:
later compromises don't reveal previous sessions.

Why?

Diffie-Hellman Protocol



Hellman and Diffie

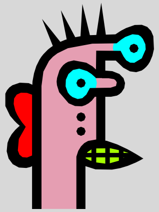
Alice and Bob never met and share no secrets

Public info: p and g

- p is a large prime number, g is a generator of Z_p^*,
- $Z_p^*=\{1, 2 \dots p-1\}$; $\forall a \in Z_p^*\ \exists i$ such that $a=g^i \bmod p$

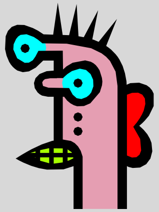
Pick secret, random X

Pick secret, random Y



$g^x \bmod p$

$g^y \bmod p$



Alice

Bob

Compute $k=(g^y)^x=g^{xy} \bmod p$

Compute $k=(g^x)^y=g^{xy} \bmod p$

Why Is Diffie-Hellman Secure?

Discrete Logarithm (DL) problem: given g^x mod p, hard to extract x

- There is no known efficient algorithm for doing this
- This is not enough for Diffie-Hellman to be secure!

Computational Diffie-Hellman (CDH) problem: given g^x and g^y, hard to compute g^{xy} mod p

- … unless you know x or y, in which case it's easy

Decisional Diffie-Hellman (DDH) problem:

given g^x and g^y, hard to tell the difference between g^{xy} mod p and g^r mod p where r is random

Properties of Diffie-Hellman

Assuming the DDH problem is hard, Diffie-Hellman protocol is a secure key establishment protocol against <u>passive</u> attackers

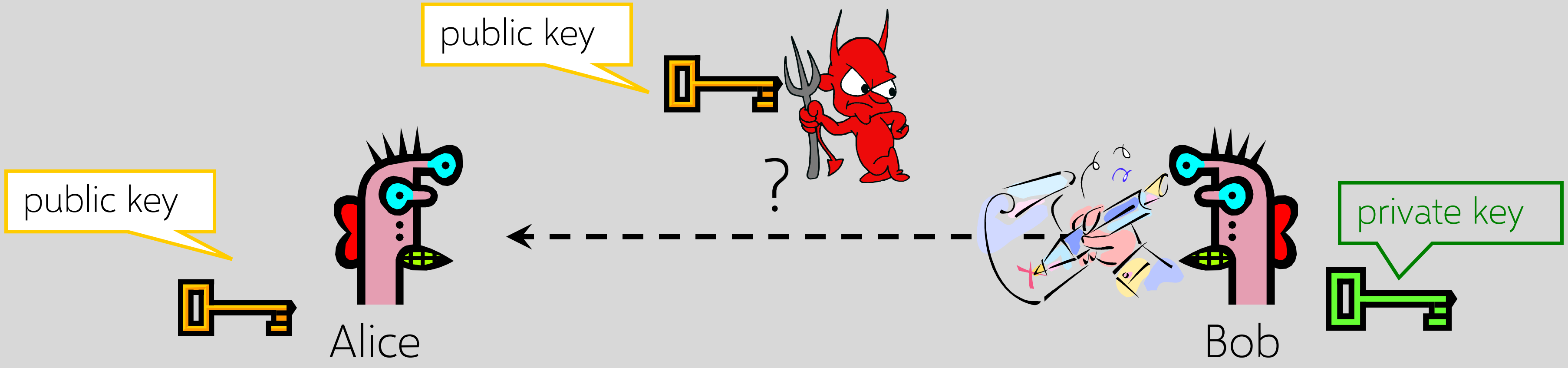
- Eavesdropper can't tell the difference between the established key and a random value
- Can use the new key for symmetric cryptography

Need an authentication mechanism in addition to Diffie-Hellman

- Examples: TLS, IPsec

Modern implementations (eg, Signal and WhatsApp) use Elliptic-Curve Diffie-Hellman

Digital Signatures: Basic Idea



Given: Everybody knows Bob's public key
Only Bob knows the corresponding private key

Goal: 1. To compute a signature on a message, must know the private key
2. To verify a signature, only need the public key (anyone can verify)

RSA Signatures

Public key is (n,e), private key is d

To **sign** message m: $s = hash(m)^d \bmod n$

- Signing and decryption are the same mathematical operation in RSA

To **verify** signature s on message m: $s^e \bmod n = (hash(m)^d)^e \bmod n = hash(m)$

- Verification and encryption are the same mathematical operation in RSA

Message must be hashed and padded (why?)

Digital Signature Algorithm (DSA)

U.S. government standard (1991-94)

- Modification of the ElGamal signature scheme (1985)

Key generation:

- Generate large primes p, q such that q divides p-1
  - $2^{159} < q < 2^{160}$, $2^{511+64t} < p < 2^{512+64t}$ where $0 \leq t \leq 8$
- Select $h \in Z_p^*$ and compute $g = h^{(p-1)/q} \bmod p$
- Select random x such $1 \leq x \leq q-1$, compute $y = g^x \bmod p$

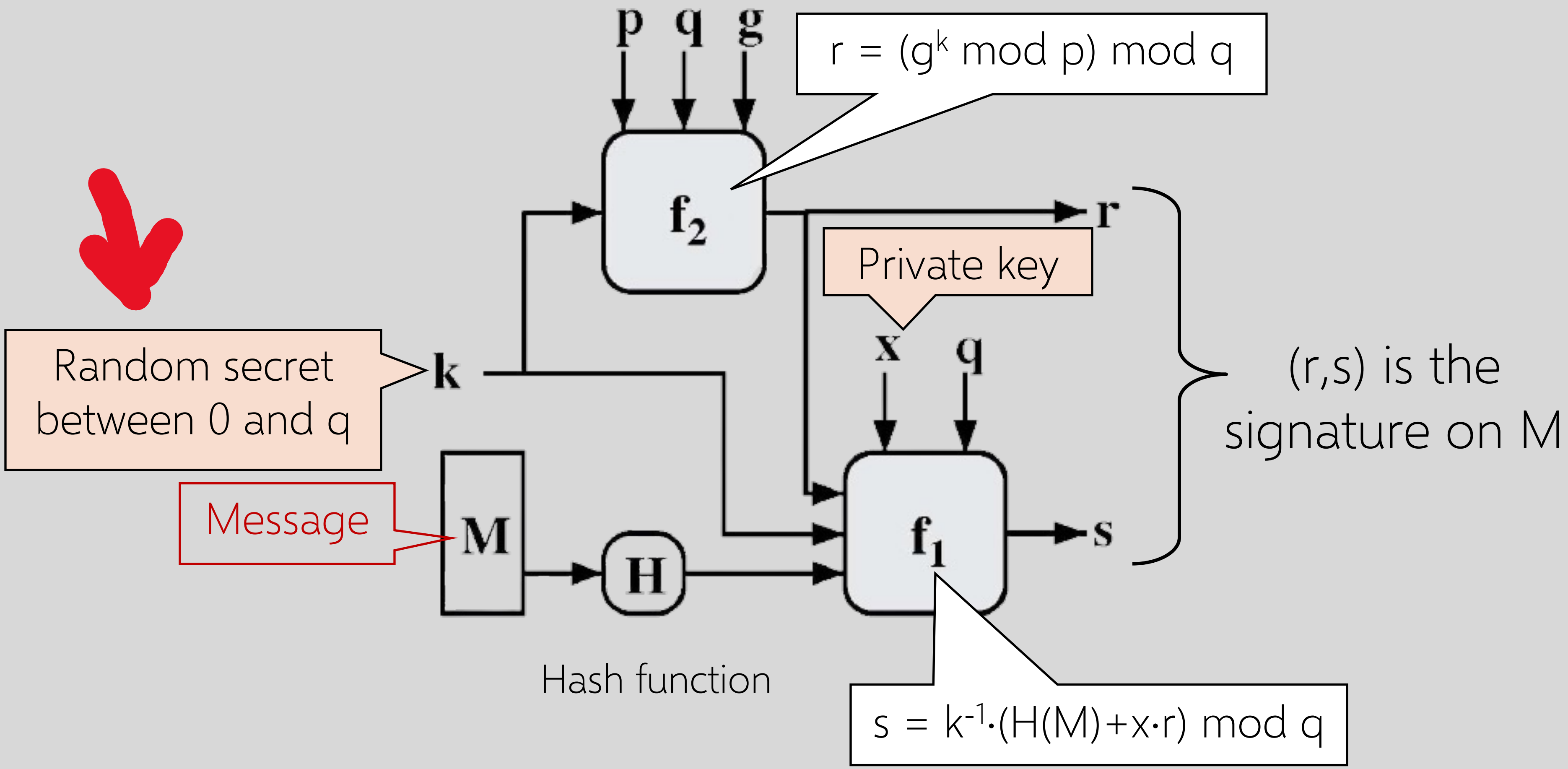
Public key: $(p, q, g, g^x \bmod p)$, private key: x

Security of DSA requires hardness of discrete log

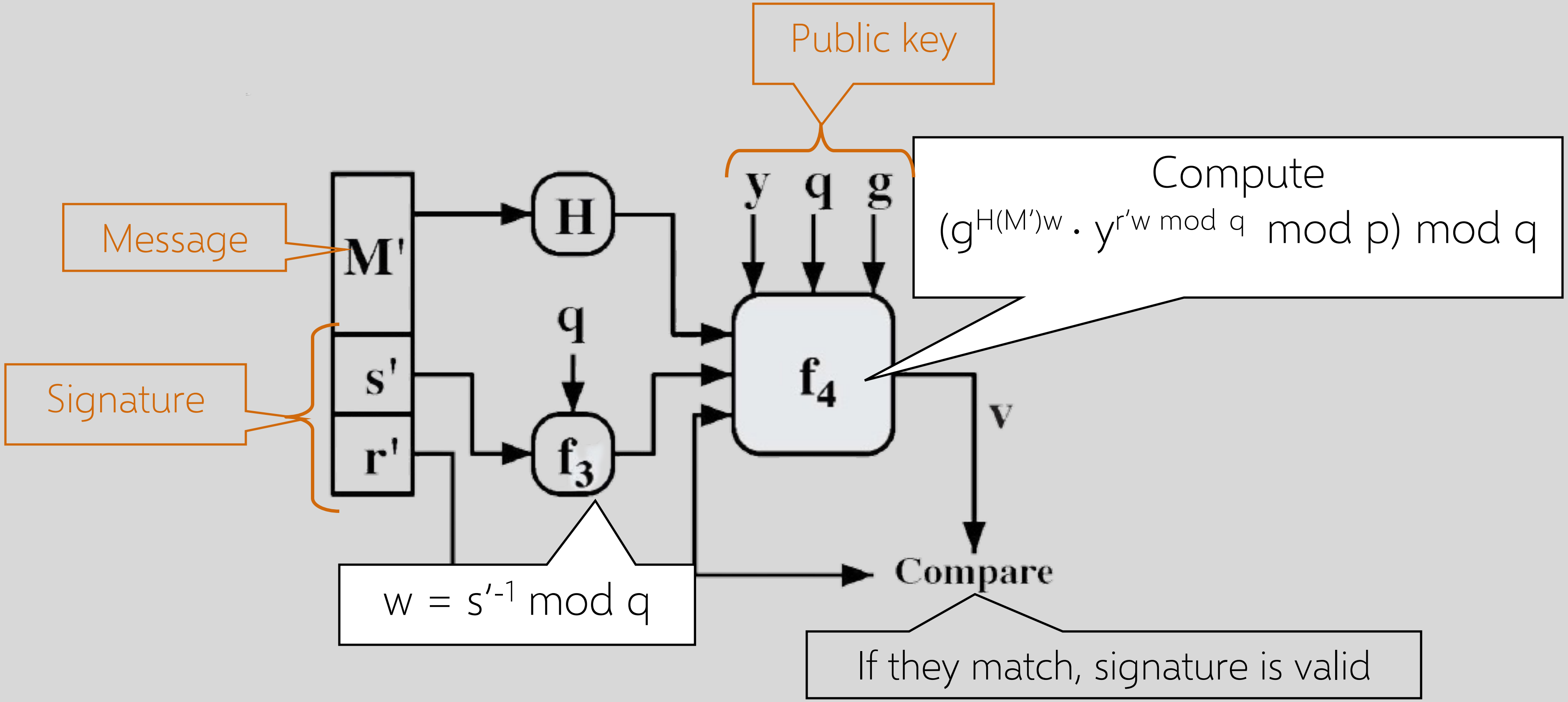
- If one can take discrete logarithms, then can extract x (private key) from $g^x \bmod p$ (public key)

Modern implementations use elliptic-curve cryptography (ECDSA)

DSA: Signing a Message



DSA: Verifying a Signature



Why DSA Verification Works

- If (r,s) is a valid signature, then $r \equiv (g^k \bmod p) \bmod q$; $s \equiv k^{-1} \cdot (H(M)+x \cdot r) \bmod q$
- Thus $H(M) \equiv -x \cdot r + k \cdot s \bmod q$
- Multiply both sides by $w = s^{-1} \bmod q$, obtain $H(M) \cdot w + x \cdot r \cdot w \equiv k \bmod q$
- Exponentiate g to both sides, obtain $(g^{H(M) \cdot w + x \cdot r \cdot w} \equiv g^k) \bmod p \bmod q$
- In a valid signature, $g^k \bmod p \bmod q = r$, $g^x \bmod p = y$
- Verify $g^{H(M) \cdot w} \cdot y^{r \cdot w} \equiv r \bmod p \bmod q$

Security of DSA

Standard security requirements for any digital signature scheme

Can't create a valid signature without private key

Can't change or tamper with signed message

If the same message is signed twice, signatures are different

- Each signature is based in part on random secret k

Random secret k must be different for each signature!

- If k is leaked or if two messages re-use the same k, attacker can recover the private key and forge any signature from then on

PS3 Epic Fail

Sony used ECDSA (DSA on elliptic curves) to sign authorized software for Playstation 3 … with the same random value in every signature

Trivial to extract master signing key and sign any homebrew software – perfect "jailbreak" for PS3 (Dec 2010)

Q: Why didn't Sony just revoke the key?

George "Geohot" Hotz

How I Hacked my Car

2022-05-22 :: greenluigi1

The OS for the infotainment system is D-Audio2V by Hyundai Mobis, some of its source code is available

2021 Hyundai Ioniq SEL

While looking through the source code avilable from Mobis’s site, I searched for all files that were shell scripts. In the results I found a shell script file called linux_envsetup.sh.

Turns out I had the zip password for the system update on my hard drive the entire time.

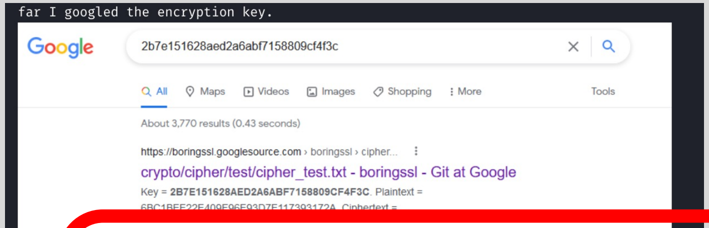
```
    cd ${UPDATE_DIR}
    zip -P ahqltmTkrhk2018@@ -r ${TARGET_DIR}/${ENC_UPDATE_PACKAGE} ./*
```

And helpfully the encryption method, key, and IV was also in the script.

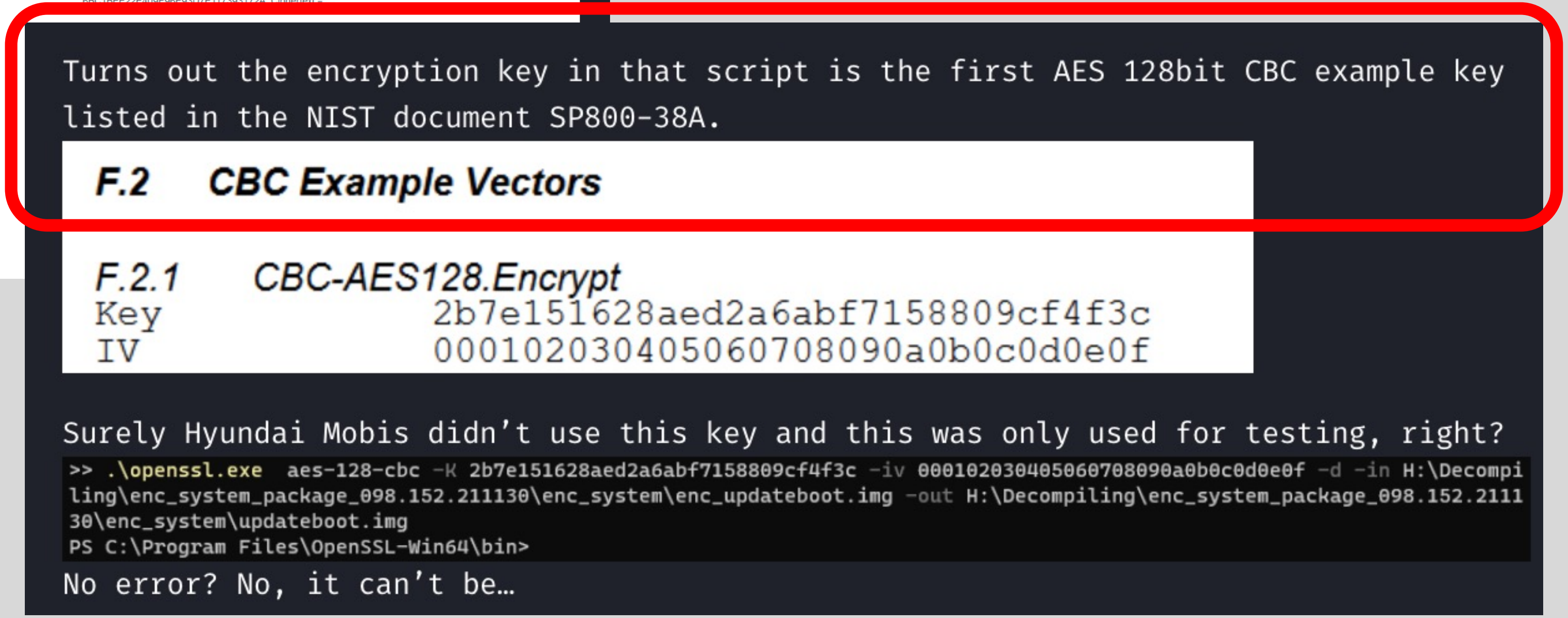
```
function generate_aes128_encryption()
{
    local FILE=$1
    local DIR=$2

    openssl aes-128-cbc -K 2b7e151628aed2a6abf7158809cf4f3c -iv 000102030405060708090a0b0c0d0e0f -e -in ${DIR}/${FILE} -out ${DIR}/enc_${FILE}
}
```

https://programmingwithstyle.com/posts/howihackedmycar/

Where Do Keys Come From?



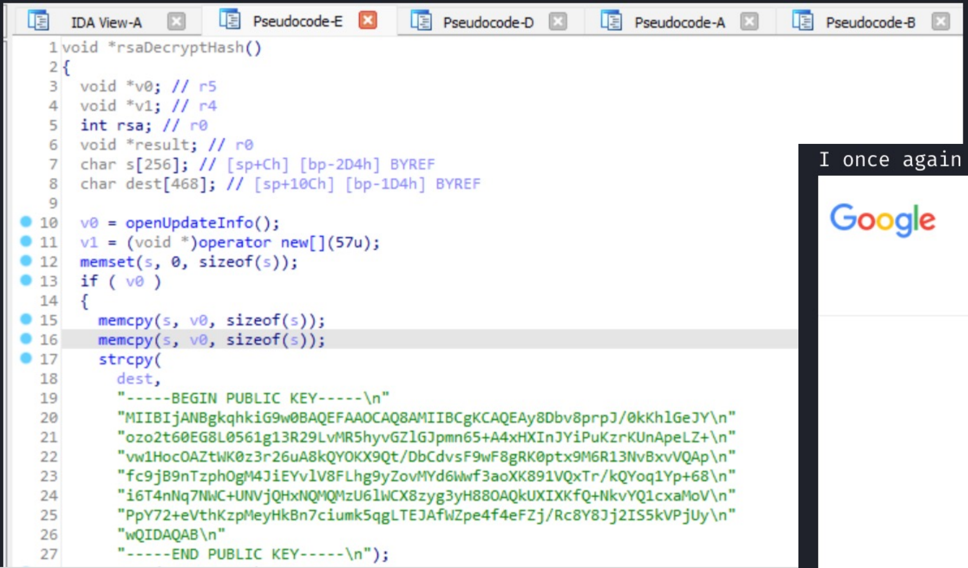
Googling the key…



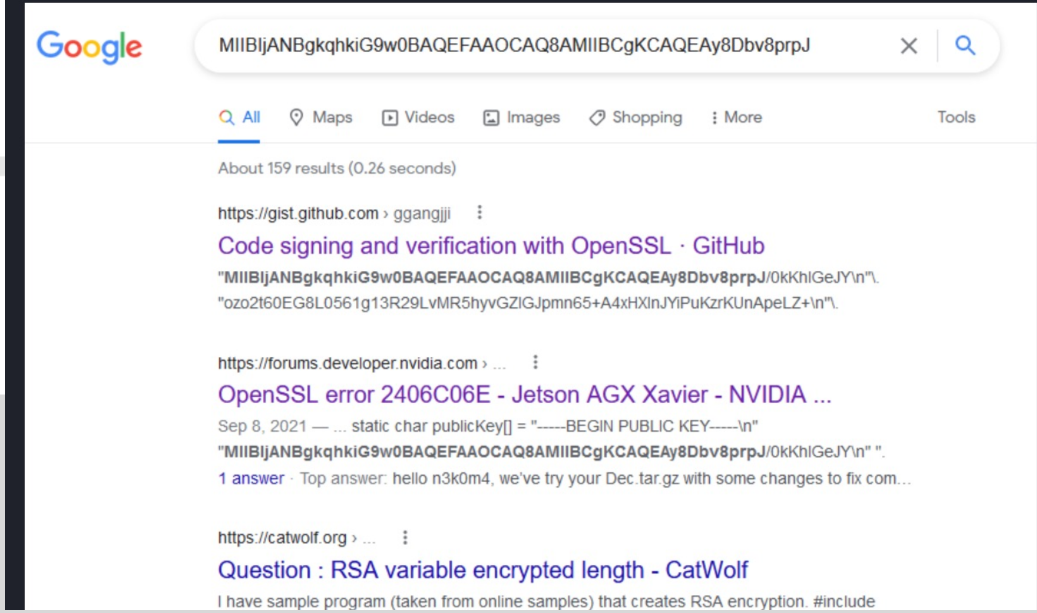
https://programmingwithstyle.com/posts/howihackedmycar/

What About Public Keys?

After searching for some keywords like “RSA” I found the public key, but no private key.



I once again googled a part of the private key as a sanity check.



https://programmingwithstyle.com/posts/howihackedmycar/

RSA Encryption & Decryption Example with OpenSSL in C

by Ravishanker Kusuma in Coding Mar 19th 2014 · 47 Comments

```c
int main(){

  char plainText[2048/8] = "Hello this is Ravi"; //key length : 2048

 char publicKey[]="-----BEGIN PUBLIC KEY-----\n"\
"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAy8Dbv8prpJ/0kKhlGeJY\n"\
"ozo2t60EG8L0561g13R29LvMR5hyvGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+\n"\
"vw1HocOAZtWK0z3r26uA8kQYOKX9Qt/DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQAp\n"\
"fc9jB9nTzphOgM4JiEYvlV8FLhg9yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68\n"\
"i6T4nNq7NWC+UNVjQHxNQMQMzU6lWCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoV\n"\
"PpY72+eVthKzpMeyHkBn7ciumk5qgLTEJAfWZpe4f4eFZj/Rc8Y8Jj2IS5kVPjUy\n"\
"wQIDAQAB\n"\
"-----END PUBLIC KEY-----\n";

 char privateKey[]="-----BEGIN RSA PRIVATE KEY-----\n"\
"MIIEowIBAAKCAQEAy8Dbv8prpJ/0kKhlGeJYozo2t60EG8L0561g13R29LvMR5hy\n"\
"vGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+vw1HocOAZtWK0z3r26uA8kQYOKX9\n"\
"Qt/DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQApfc9jB9nTzphOgM4JiEYvlV8FLhg9\n"\
"yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68i6T4nNq7NWC+UNVjQHxNQMQMzU6l\n"\
"WCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoVPpY72+eVthKzpMeyHkBn7ciumk5q\n"\
"gLTEJAfWZpe4f4eFZj/Rc8Y8Jj2IS5kVPjUywQIDAQABAoIBADhg1u1Mv1hAAlX8\n"\
"omz1Gn2f4AAW2aos2cM5UDCNw1SYmj+9SRIkaxjRsE/C4o9sw1oxrg1/z6kajV0e\n"\
"N/t008FdlVKHXAIYWF93JMoVvIpMmT8jft6AN/y3NMpivgt2inmmEJZYNioFJKZG\n"\
"X+/vKYvsVISZm2fw8NfnKvAQK55yu+GRWBZGOeS9K+LbYvOwcrjKhHz66m4bedKd\n"\
"gVAix6NE5iwmjNXktSQlJMCjbtdNXg/xo1/G4kG2p/MO1HLcKfe1N5FgBiXj3Qjl\n"\
"vgvjJZkh1as2KTgaPOBqZaP03738VnYg23ISyvfT/teArVGtxrmFP7939EvJFKpF\n"\
"1wTxuDkCgYEA7t0DR37zt+dEJy+5vm7zSmN97VenwQJFWMiulkHGa0yU3lLasxxu\n"\
"m0oUtndIjenIvSx6t3Y+agK2F3EPbb0AZ5wZ1p1IXs4vktgeQwSSBdqcM8LZFDvZ\n"\
"uPboQnJoRdIkd62XnP5ekIEIBAfOp8v2wFpSfE7nNH2u4CpAXNSF9HsCgYEA2l8D\n"\
"JrDE5m9Kkn+J4l+AdGfeBL1igPF3DnuPoV67BpgiaAgI4h25UJzXiDKKoa706S0D\n"\
"4XB74zOLX11MaGPMIdhlG+SgeQfNoC5lE4ZWXNyESJH1SVgRGT9nBC2vtL6bxCVV\n"\
"WBkTeC5D6c/QXcai6yw6OYyNNdp0uznKURe1xvMCgYBVYYcEjWqMuAvyferFGV+5\n"\
"nWqr5gM+yJMFM2bEqupD/HHSLoeiMm2O8KIKvwSeRYzNohKTdZ7FwgZYxr8fGMoG\n"\
"PxQ1VK9DxCvZL4tRpVaU5Rmknud9hg9DQG6xIbgIDR+f79sb8QjYWmcFGc1SyWOA\n"\
"SkjlykZ2yt4xnqi3BfiD9QKBgGqLgRYXmXp1QoVIBRaWUi55nzHg1XbkWZqPXvz1\n"\
"I3uMLv1jLjJlHk3euKqTPmC05HoApKwSHeA0/gOBmg404xyAYJTDcCidTg6hlF96\n"\
"ZBja3xApZuxqM62F6dV4FQqzFX0WWhWp5n301N33r0qR6FumMKJzmVJ1TA8tmzEF\n"\
"yINRAoGBAJqioYs8rK6eXzA8ywYLjqTLu/yQSLBn/4ta36K8DyCoLNlNxSuox+A5\n"\
"w6z2vEfRVQDq4Hm4vBzjdi3QfYLNkTiTqLcvgWZ+eX44ogXtdTDO7c+GeMKWz4XX\n"\
"uJSUVL5+CVjKLjZEJ6Qc2WZLl94xSwL71E41H4YciVnSCQxVc4Jw\n"\
"-----END RSA PRIVATE KEY-----\n";
```

http://hayageek.com/rsa-encryption-decryption-openssl-c/

Using Cryptography

Don’t roll your own!

Don’t try to implement cryptographic algorithms

Do generate your own random keys!

Do use standard libraries and APIs… correctly!