# SVGS Computer Security, Cryptology Test

My motive in writing this test is simple, to get students to review the slides. This test is open book. The questions are in the same order that the slides are in.

1. Why did WEP fail?

One or more of the following:  
Initialization Vectors were reused (key space too small)  
RC4 stream cipher used improperly (no random IV, did not drop beginning of stream)  
Allows attackers to inject any packets they wanted (No message integrity checking)

1. What is wrong with this statement? “I used base64 encryption to hide my data.”

Base64 is encoding, not encryption. Many programs are available to decode base64

1. How is the modular inverse of a number usually calculated?

Use Euclid’s Extended Algorithm (they may also say "import inverse from pycryptodome")

1. What is the modular inverse of 5 mod 13 ?

8

1. What is the most important thing to remember about an IV or nonce?

It can never be reused

1. What is wrong with AES-ECB mode?

Repeated blocks of plaintext give the same blocks of cipher text. See the Linux penguin picture in the slides.

1. What advantage does symmetric key encryption have over asymmetric key encryption? What disadvantage?

Symmetric encryption is much faster than asymmetric.  
Symmetric encryption does not give an easy way to exchange keys.

1. What is non-repudiation?

The sender cannot claim they didn’t send/sign the message. In public key encryption it was signed with the user’s private key, which only the user has.

1. Why is RSA slow when you use it to encrypt a large file?

One or more of the following:  
Each block must be encrypted/decrypted by taking a large number to a large power in modular arithmetic. (Slow.)  
The block size is only as big as the key size.

1. With RSA encryption, if Alice sends a message to Bob, she encrypts the message with…

Bob’s public key

1. How do computers get the large prime numbers they use in encryption? How do they know they are prime?

They choose their large prime numbers randomly, and then test with a primality test (Miller-Rabin). Repeat until a prime is found.

1. Why is a good PRNG important in cryptography?

If your keys are not truly random, it makes it much easier for an attacker to break them.

1. What is the most popular Key exchange method for HTTPS? For SSH?

Last DH slide, HTTPS most popular is ECDH, SSH is (almost a tie) DH

1. Is Elliptic Curve Cryptography more closely related to Diffie-Hellman or to RSA cryptography?

ECC is an extended digital logarithm problem, and DH is a digital logarithm problem. DH.

1. What are the public portions of a key exchange using Elliptic Curve Cryptography?

Curve, modulus, base point, Alice and Bob’s public keys

1. What is the primary difference between a digital signature and a MAC?

A signature is based on public key encryption, so non-repudiation applies. MACs are based on symmetric keys; non-repudiation does not apply.

1. What are the two properties a cryptographic hash is supposed to have?

One-way  
Collision resistance

1. A Digital Certificate is used to distribute a …

Public Key

1. Why is it important to know where a public key really came from?

If you use a public key provided by an attacker, she can use an MITM attack against you.

1. When a browser uses HTTPS, how does it know it is talking to the correct server and not an imposter?

It verifies the server’s digital certificate

1. A browser and a server agree on a cipher suite instead of just saying ‘TLS’. What are the different protocols in the suite used for?

The different protocols are used for:  
Key Exchange  
Digital Certificates  
Symmetric encryption  
Message authentication/integrity checking  
Hashing

1. In TLS the network traffic is encrypted, and the school’s/company’s IPS cannot inspect the data to see if it is malware. What information is available (at least in TLS v1.2 and below) that can give some clues to network security personnel?

The digital certificates used in the TLS handshake will give the analyst basic information about where the traffic is going, even though they cannot decrypt the traffic.

1. What key exchange protocols may be used for ‘Perfect’ Forward Secrecy? What protocol may not be used?

Diffie-Hellman Key Exchange (DHKE) and Elliptic Curve Cryptography (ECC) may be used because private key, public key and session key is computed separately for each session. RSA may not be used because only the server’s private key is used and it does not change; if the server’s private key is available, all current and stored network traffic encrypted with RSA may be decrypted.

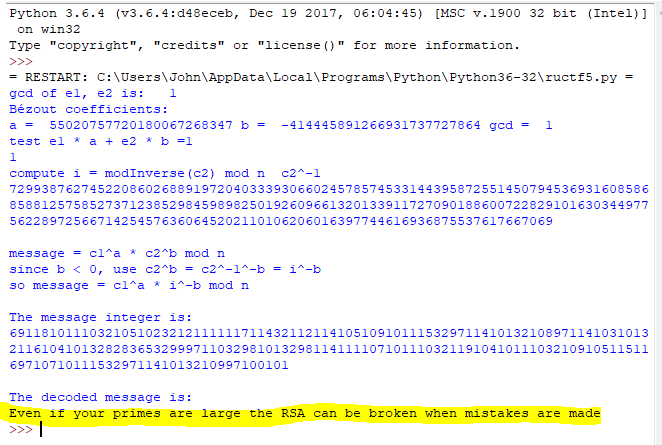
BONUS QUESTION

This tells you what you need to know:  
<https://crypto.stackexchange.com/questions/1614/rsa-cracking-the-same-message-is-sent-to-two-different-people-problem>   
You will also need to compute the modular inverse of a number and find the Bézout coefficients of two numbers. The Bézout coefficients are computed in the Extended Euclidean Algorithm we use to compute modular inverses but not displayed. The coefficients are in the u1 and u2 variables in findModInverse. (I did it in Python, but there is undoubtedly a web site that will compute the Bézout coefficients for you.) Once you have those coefficients, the problem is simple. There is one twist, in that the Bézout coefficients usually have opposite signs. That’s why he adds a step, where i = c2^(-1) mod n, ie i = modular inverse of c2. That way he doesn’t have to take a number to a negative power mod n.

Note: When you decode the message, they use a strange way of putting ASCII into the message integer. It is ASCII in decimal and then strung together. For example, "abc def" would be encoded 97989932100101102

Here is the question from the RU CTF.

Suppose you intercept the ciphertext integer  
[3348898614019888901908403254933640035246212844961838831090198209986980794381721549119817432681384755697410365192319104804164099565189541033745677177908430898159000425144231080271489963068735078029094600990872466478190741629812176]  
that encrypted a plaintext message with an ASCII alphabet assignment using the RSA cryptosystem with public encryption exponent of  
e = 927497329847987298271115  
and modulus  
m = 4035789025935566763434217693291904203514985559759202218772232737779637242777118595044390460183072421339720558176591333566629680159420540355202801063004396853930869779589477542063791290354739283500845851153515283182096350655220153  
  
You also intercepted the ciphertext integer  
[239640052909589767377717332389707447467040807634556900631678847178249840124011908871933438491107613018104449557989400002181849114477870950132116542696807901617211160049032412890334084433427701567750018959947232564370889351855337]  
that represented an encryption of the same plaintext using the RSA cryptosystem with public encryption exponent of  
  
e = 123132131231124141411111 and modulus  
m = 4035789025935566763434217693291904203514985559759202218772232737779637242777118595044390460183072421339720558176591333566629680159420540355202801063004396853930869779589477542063791290354739283500845851153515283182096350655220153  
  
Decipher the message.



c1 = 3348898614019888901908403254933640035246212844961838831090198209986980794381721549119817432681384755697410365192319104804164099565189541033745677177908430898159000425144231080271489963068735078029094600990872466478190741629812176

e1 = 927497329847987298271115

n = 4035789025935566763434217693291904203514985559759202218772232737779637242777118595044390460183072421339720558176591333566629680159420540355202801063004396853930869779589477542063791290354739283500845851153515283182096350655220153

c2 = 239640052909589767377717332389707447467040807634556900631678847178249840124011908871933438491107613018104449557989400002181849114477870950132116542696807901617211160049032412890334084433427701567750018959947232564370889351855337

e2 = 123132131231124141411111

#https://crypto.stackexchange.com/questions/1614/rsa-cracking-the-same-message-is-sent-to-two-different-people-problem

from Crypto.Util.number import GCD, inverse

b = GCD(e1, e2)

print("gcd of e1, e2 is: ",b)

#Extended Euclidean Algorithm, compute a, b

#stolen from FindModInverse in https://inventwithpython.com/cracking/

# e1 \* a + e2 \* b = gcd(e1, e2) = 1

u1, u2, u3 = 1, 0, e1

v1, v2, v3 = 0, 1, e2

while v3 != 0:

q = u3 // v3

v1, v2, v3, u1, u2, u3 = (u1 - q \* v1), (u2 - q \* v2), (u3 - q \* v3), v1, v2, v3

a, b = u1, u2

print("Bézout coefficients:")

print("a = ", a,"b = ", b, "gcd = ", u3)

print("test e1 \* a + e2 \* b =1")

print(e1\*a + e2\*b)

print ("compute i = inverse(c2) mod n, or c2^-1")

i =inverse(c2, n)

print(i)

print("\nmessage = c1^a \* c2^b mod n")

print("since b < 0, use c2^b = c2^-1^-b = i^-b")

print("so message = c1^a \* i^-b mod n\n")

message = pow(c1, a, n) \* pow(i, -b, n) % n

print("The message integer is:")

print(message)

print("\nThe decoded message is:")

q = message

flag = []

while q > 0:

qmod = q % 100

if qmod < 30:

flag.insert(0,chr(q % 1000))

q = q // 1000

else:

flag.insert(0,chr(qmod))

q = q // 100

print(''.join(flag))

gcd of e1, e2 is: 1

Bézout coefficients:

a = 55020757720180067268347 b = -414445891266931737727864 gcd = 1

test e1 \* a + e2 \* b =1

1

compute i = Inverse(c2) mod n c2^-1

729938762745220860268891972040333930660245785745331443958725514507945369316085868588125758527371238529845989825019260966132013391172709018860072282910163034497756228972566714254576360645202110106206016397744616936875537617667069

message = c1^a \* c2^b mod n

since b < 0, use c2^b = c2^-1^-b = i^-b

so message = c1^a \* i^-b mod n

The message integer is:

691181011103210510232121111117114321121141051091011153297114101321089711410310132116104101328283653299971103298101329811411110710111032119104101110321091051151169710710111532971141013210997100101

The decoded message is:

Even if your primes are large the RSA can be broken when mistakes are made