Lab 2: Exploring Public Key Cryptography

Revised: January 23, 2018

Please read through the whole lab before starting it.

Background. In this lab, you will explore asymmetric key (public key) cryptography by implementing the Diffie Hellman key exchange protocol and RSA encryption scheme. You will explore the properties of these schemes, and see how naive implementations can lead to insecurity. You will also learn to use GPG, a commonly used set of cryptographic tools used for secure communication.

Environment. You are free to implement the solutions to this lab using the programming language and cryptographic library of your choosing. However, the instructor's strong recommendation is that you use Python and PyCrypto (or some other high-level language with decent crypto support).

Task I. Implement Diffie Hellman Key Exchange

The goal of this task is to get your public key crypto juices flowing by implementing one of the single most important discoveries in modern cryptography: the Diffie Hellman Key Exchange. In a single program, emulate the following protocol:

Alice		**Bob**	
Privately Computes	**Sends**	**Privately Computes**	**Sends**
	p,g		
Pick random element $a \in \mathbb{Z}_p$ $A = g^a \bmod p$		Pick random element $b \in \mathbb{Z}_p$ $B = g^b \bmod p$	
	A		B
$s = B^a \bmod p$		$s = A^b \bmod p$	
k = SHA256(s)		k = SHA256(s)	
m_0 = "Hi Bob!" $c_0 = \text{AES-CBC}_k(m_0)$	c_0	m_1 = "Hi Alice!" $c_1 = \text{AES-CBC}_k(m_1)$	c_1

Try it in a small group first, setting p = 37 and g = 5. Confirm Alice and Bob compute the same symmetric key k. Truncate the output of SHA256 to 16 bytes, so that you can use it to AES-CBC encrypt some messages between Alice and Bob.

How hard would it be for an adversary to solve the Diffie Hellman Problem (DHP) given these parameters? What strategy might the adversary take?

Now, let's use some "real life" numbers. IETF suggests the following 1024-bit parameters[1]:

```
p =
B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6
9A6A9DCA 52D23B61 6073E286 75A23D18 9838EF1E 2EE652C0
13ECB4AE A9061123 24975C3C D49B83BF ACCBDD7D 90C4BD70
98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F F55BCCC0
A151AF5F 0DC8B4BD 45BF37DF 365C1A65 E68CFDA7 6D4DA708
DF1FB2BC 2E4A4371

g =
A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F
D6406CFF 14266D31 266FEA1E 5C41564B 777E690F 5504F213
160217B4 B01B886A 5E91547F 9E2749F4 D7FBD7D3 B9A92EE1
909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01 69B6A28A
D662A4D1 8E73AFA3 2D779D59 18D08BC8 858F4DCE F97C2A24
855E6EEB 22B3B2E5
```

Modify your implementation to use these parameters, make sure Alice and Bob can compute the same symmetric key k, and correctly exchange some encrypted messages.

Would the same strategy used for the tiny parameters work here? Why or why not?

Task II. Implement MITM Key Fixing & Negotiated Groups

Diffie-Hellman, as written above, is only secure against a passive (eavesdropping) adversary. As you will soon see, very bad things can happen if the adversary is able to tamper with the numbers in the protocol.

---

[1] Here are some ways to get these numbers into Python:

```
p =
int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D18
9838EF1E2EE652C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD7098488E9C219A73724EFFD
6FAE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708DF1FB2BC2E
4A4371", 16)
q =
int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564B
777E690F5504F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1909D0D2263F80A76A6A24
C087A091F531DBF0A0169B6A28AD662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24855E6EEB22
B3B2E5", 16)
```

A. Modify your implementation from Task I in the following way:

Alice		Mallory	Bob	
Computes	**Sends**	**Modifies**	**Computes**	**Sends**
	p,g			
$a \in \mathbb{Z}_p$ $A = g^a \bmod p$			$b \in \mathbb{Z}_p$ $B = g^b \bmod p$	
	A	A→p		
		B→p		B
$s = B^a \bmod p$			$s = A^b \bmod p$	
k = SHA256(s)			k = SHA256(s)	
m_0 = "Hi Bob!" $c_0 = \text{AES-CBC}_k(m_0)$	c_0		m_1 = "Hi Alice!" $c_1 = \text{AES-CBC}_k(m_1)$	c_1

In your implementation, show that Mallory can successfully decrypt the messages c_0 and c_1.

B. Repeat this attack, but instead of tampering with A and B, tamper with the generator g. Show that Mallory can recover some of the messages by setting g to 1, p, or p-1.

Why were these attacks possible? What is necessary to prevent them?

Task III. Implement "textbook" RSA & MITM Key Fixing via Malleability

From Diffie Hellman, we move to the next monumental breakthrough in public key cryptography: RSA, named for its inventors: Rivest, Shamir and Adelman.

A. Let's implement it! RSA has two core components: key generation and encryption/decryption. (90% of the work is in implementing key generation.) Your implementation should support variable length primes (up to 2048 bits), and use the value e = 65537. Feel free to use your cryptographic library's interface for generating large primes, but implement the rest–including computing the multiplicative inverse–yourself.

Encrypt and decrypt a few messages to yourself to make sure it works. Remember messages must be integers in $\mathbb{Z}^*_n$. You can convert an ASCII string to hex, and then turn that hex value into an integer.

While it's very common for many people to share an e (common values are 3,7, 2^{16}+1), it is very bad if two people share an RSA modulus n*. Briefly describe why this is, and what the ramifications are.*

B. What you just implemented is often called "textbook" RSA, and it is wildly insecure. Because it is too slow and inconvenient to operate on a large amount data directly, RSA is often used to exchange a symmetric key that will be used to encrypt future messages. It would be terrible, of course, if an adversary were able to learn that key. And, that's what we're about to do.

One of textbook RSA's great weaknesses is its malleability, *i.e.* an active attacker can change the meaning of the plaintext message by performing an operation on the respective ciphertext. To demonstrate the dangers of malleability, implement the following protocol:

Alice		**Mallory**	**Bob**	
Computes	**Sends**	**Modifies**	**Computes**	**Sends**
	n,e			
			$s \in \mathbb{Z}^*_n$ $c = s^e \bmod n$	
				c
		c' = F(c)		
$s = c'^d \bmod n$				
k = SHA256(s)				
m = "Hi Bob!" $c_0 = \text{AES-CBC}_k(m)$	c_0			

Find the operation F(·) that Mallory needs to apply to the ciphertext c that will allow her to decrypt the ciphertext c_0. HINT: Mallory knows Alice's public key (n,e) and can encrypt her own messages to Alice.

Give another example of how RSA's malleability could be used to exploit a system (e.g. to cause confusion, disruption, or violate integrity).

Malleability can also affect signature schemes based on RSA. Consider the scheme:

$\text{Sign}(m,d) = m^d \bmod n$

Suppose Mallory sees the signatures for two message m_1 and m_2. Show how Mallory can create a valid signature for a third message, $m_3 = m_1 \cdot m_2$.

C. Forward Secrecy

Forward Secrecy (sometimes called Perfect Forward Secrecy) is a property of key-agreement protocols that ensures that all prior communications encrypted with a symmetric session key, derived from a long-term private key, will still remain secure, even if the private key becomes compromised. However, not all key exchange protocols exhibit forward secrecy. *Briefly justify whether either of the following key exchange protocols do or do not provide forward secrecy.* HINT: You need not worry about active adversaries (*i.e.* man-in-the-middle or message tampering attacks). Instead, consider only an eavesdropping adversary that keeps a transcript of all traffic between Alice and Bob, but may later be able to compel Alice or Bob to give up their secrets.

Alice		**Bob**	
Privatle Computes	**Sends**	**Privately Computes**	**Sends**
	p,g		
$a \in \mathbb{Z}_p$ $A = g^a \bmod p$		$b \in \mathbb{Z}_p$ $B = g^b \bmod p$	
	A		B
$s = B^a \bmod p$ Forget a		$s = A^b \bmod p$ Forget b	
$k = SHA256(s)$		$k = SHA256(s)$	
m_0 = “Hi Bob!” $c_0 = AES\text{-}CBC_k(m_0)$	c_0	m_1 = “Hi Alice!” $c_1 = AES\text{-}CBC_k(m_1)$	c_1

Protocol 1: A Diffie-Hellman key-agreement

Alice		**Bob**	
Privately Computes	**Sends**	**Privately Computes**	**Sends**
RSA key pair $<A_{pub},A_{pri}>$			
	A_{pub}		
		Pick random element $x \in \mathbb{Z}^*_n$ $y = RSA(A_{pub},x)$	
			y
$x = RSA^{-1}(A_{pri}, y)$ $k = SHA256(x)$		$k = SHA256(x)$	
m_0 = “Hi Bob!” $c_0 = AES\text{-}CBC_k(m_0)$	c_0	m_1 = “Hi Alice!” $c_1 = AES\text{-}CBC_k(m_1)$	c_1

Protocol 2: An RSA key exchange

Task IV. GPG: A Real Life Public Key Crypto Tool

For this task, you will generate your own GNU Privacy Guard (GPG) public key pair using Keybase. Before Keybase came along, most users used the command line interface to GPG. This web tutorial along with GPG's MiniHOWTO might show you how bad that actually is.

A. If you don't already have a public key, visit https://keybase.io, create an account, and generate a public key pair for yourself. Feel free to use the default parameters, or play with them as you see fit. If possible, verify yourself using a few methods. The idea being: the more methods authenticated methods you use, the more trust you build in the authenticity of your key.

B. Now, join the web of trust by "following" some of your friends, classmates, or me. Optionally, you can use an HKP server, such as `keys.gnupg.net`, to publish and exchange keys. Be sure to verify the identity of the owner and key before signing it. Verifying the key's fingerprint with the owner over a "secure" or out-of-band channel (e.g. face-to-face, over the phone, or by trusting keybase's authentication techniques) is an easy way to do this.

C. Finally, encrypt and sign a message to Prof. Peterson, and publish it as a plaintext attachment to the Secret Message discussion topic within the CPE321 Student Forum on PolyLearn. Feel free to use keybase's web or command-line interface to do this. For verification, here is his key fingerprint:

```
BC57A618547771A977BFBD14F94EFBE0D5E6E6CB
```

D. (Optional) If you don't want to use your key anymore you can destroy your keybase account.

Task V. While the features provided by public key cryptography are attractive, they come at the cost of performance. In this task, you will quantify the performance differences between public and symmetric key algorithms. Fortunately, OpenSSL provides a simple interface for doing so.

```
openssl speed rsa
```

and

```
openssl speed aes
```

will perform and measure their respective public and symmetric key operations using different parameters. One of the returned results for both operations is a measure of throughput: operations per time (e.g. signatures per second). *Run these operations and report your findings. Include two graphs: one that plots the block size vs. throughput for the various AES*

modes of operations and one that plots the RSA key size vs. throughput for each RSA operation. How do the results compare?

Submission. Write a brief report describing what you did and what you observed. Include any code that you wrote, as well as answers to any questions (these are in italics), within the report itself. Please include any explanations of the surprising or interesting observations you made.

Write at a level that demonstrates your technical understanding, and do not shorthand ideas under the assumption that the reader already "knows what you mean." Think of writing as if the audience was a smart colleague who may not have taken this class.

Describe what you did in sufficient detail that it can be reproduced. Please do not use screenshots of the VM to show the commands and code you used, but rather paste (or carefully re-type) these into your report from your terminal. Submit your completed write up to PolyLearn in PDF format.