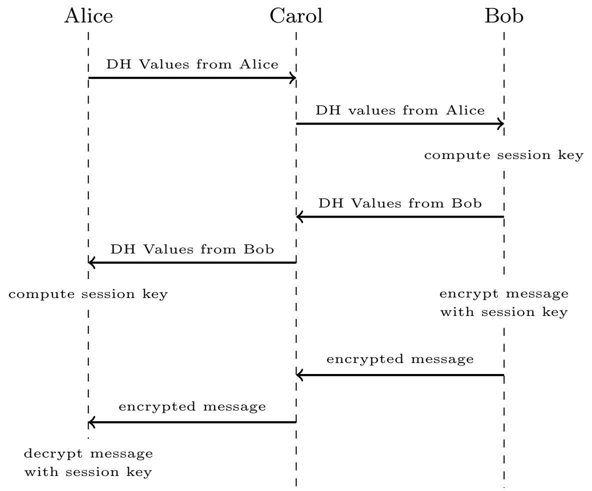
20:08 Encryption is Not Integrity!

by Cornelius Diekmann

Don't we all remember the following common setup from our introductory security course? Bob wants to send a secret message to Alice. In order to obtain a key for encrypting the message, Alice and Bob first use Diffie-Hellman (DH) to exchange a fresh session key. With this fresh session key, Bob symmetrically encrypts the message and sends it to Alice. Carol volunteers to transmit the messages between Bob and Alice. Here is the setup:



One of the first things we learn in our introductory security course is that Carol could Man-in-the-Middle (MitM) the DH exchange to obtain session keys with Alice and Bob herself, while poor Alice and poor Bob still believe they are talking privately with each other. The next thing an introductory security course teaches us is how to prevent this attack. And here is how this article differs from an introductory security course: Bob has the misconception that he can use encryption to prevent unauthorized modification. As the title suggests, this does not work out well for Bob. Neighbors, don't act like Bob.

Let us hear the story of Alice, Bob, and Carol. Bob will make five different attempts to transmit the encrypted message to Alice. He will try to use RSA encryption to prevent a MitM attack. The protocol aborts prematurely if Carol could break the key before Bob has sent the message.

I hear our quality-conscious readers ask "S-tory?", surely followed by "PoC or GTFO!" Esteemed reader, don't worry, the text you are reading right now was generated by `poc.py`[36].

"Couldn't Bob just use TLS?", you might ask. For sure! A TLS handshake would authenticate the DH values and everything would be fine. But using a ready-made TLS implementation would also be boring. Furthermore, the handshake sketched above is not TLS. In the course of this story, Bob will use parts of the OpenSSL library to do parts of the DH handshake for him. Will this help? Let the story begin.

Run 0: Prologue and Short recap of Diffie-Hellman

Alice and Carol are just returning from their introductory security course. Bob, who also attended the lecture, walks over to Alice. "If a message is encrypted, an attacker cannot read it and thus cannot modify it," Bob says to Alice. Alice knows that encryption does not provide integrity and immediately wants to call bullshit on Bob's claim. But she hesitates for a moment. Bob won't appreciate an abstract explanation anyway. "Let's see where this is going," she thinks and agrees to follow his explanation. "I hope there will be code?" Alice responds. Bob nods.

"Carol, come over, Bob is explaining crypto," Alice shouts to Carol. Bob starts explaining, "Let's first create a fresh session key so I can send a secret message to you, Alice." Alice agrees, this sounds like a good idea. To make the scenario realistic, Alice makes sure that neither Bob nor Carol can see her screen. She opens her python3 shell and is about to generate some DH values. "We need a large prime p and a generator g," Alice says. "607 is a prime", Bob says with Wikipedia open in his browser. Alice, hoping that Bob is joking about the size of his prime, suggests the smallest prime from RFC 3526 as an example:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
```

[36] `unzip pocorgtfo20.pdf poc.py` or `git clone https://github.com/diekmann/encryption-is-not-integrity.git`

```
    83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
    670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

This is a 1536-bit prime. Alice notes fascinated, "this prime has π in it!"

According to the RFC, the prime is $p = 2^{1536} - 2^{1472} - 1 + 2^{64} \cdot (\lfloor 2^{1406} pi \rfloor + 741804)$. Alice continues to think aloud, "Let me reproduce this. Does that formula actually compute the prime? Python3 integers have unlimited precision, but π is not an integer."

"Python also has floats," Bob replies. Probably Bob had not been joking when he suggested 607 as large prime previously. It seems that Bob has no idea what 'large' means in cryptography. Meanwhile, using

```
>>> import decimal
```

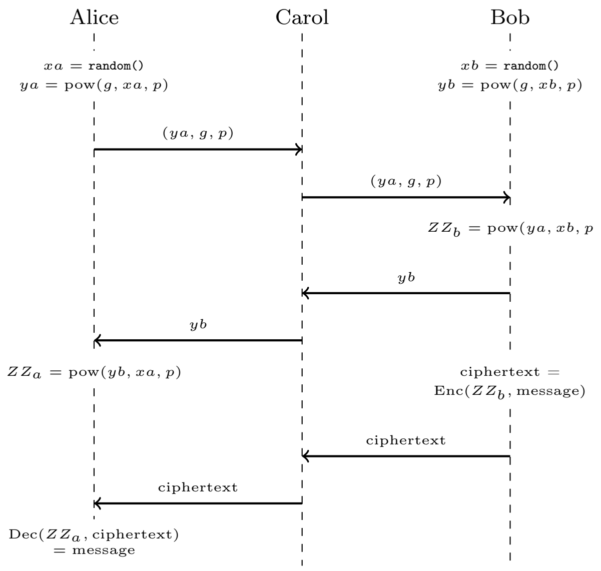
Alice has reproduced the calculation. By the way, the generator g for said prime is conveniently 2.

A small refresher on DH follows. Note that the RFC uses "^" for exponentiation.

```
=== BEGIN SNIPPET RFC 2631 ===
2.1.1.  Generation of ZZ

   [...] the shared secret ZZ is generated as follows:

     ZZ = g ^ (xb * xa) mod p

   Note that the individual parties actually perform the
   computations:

     ZZ = (yb ^ xa)  mod p  = (ya ^ xb)  mod p

   where ^ denotes exponentiation

         ya is party a's public key; ya = g ^ xa mod p
         yb is party b's public key; yb = g ^ xb mod p
         xa is party a's private key
         xb is party b's private key
         p is a large prime
=== END SNIPPET RFC 2631 ===
```

Alice takes the initiative, "Okay, I generate a secret value (xa), compute $ya = g^{xa} \bmod p$ and send to you ya, g, p. This is also how we did it in the lecture." Bob then has to choose a secret value (xb), compute $yb = g^{xb} \bmod p$ and send yb back to Alice, so she can compute ZZ_a. Bob then uses the key ZZ_b he computed to encrypt a message and send it to Alice. Since $ZZ_b = ZZ_a$, Alice can decrypt the message.

This is what Alice and Bob plan to do:



Alice
Carol
Bob
xa = random()
ya = pow(g, xa, p)
xb = random()
yb = pow(g, xb, p)
(ya, g, p)
(ya, g, p)
ZZ_b = pow(ya, xb, p
yb
yb
ZZ_a = pow(yb, xa, p)
ciphertext = Enc(ZZ_b, message)
ciphertext
ciphertext
Dec(ZZ_a, ciphertext) = message

"Let's go then," Bob says. "Wait," Alice intervenes, "DH is only secure against passive attackers. An active attacker could MitM our exchange." Alice and Bob look at Carol, she smiles. Alice continues, "What did you say in the beginning?" "Right," Bob says, "we must encrypt our DH values, so Carol cannot MitM us." Fortunately, Alice and Bob have 4096-bit RSA keys and have securely distributed their public keys beforehand.

"Okay, what should I do?" Alice asks. She knows exactly what to do, but Bob's stackoverflow-driven approach to crypto may prove useful in the course of this story. Bob types into Alice's terminal:

```
>>> import Crypto.PublicKey.RSA
>>> def RSA_enc(k_pub, msg):
...     return k_pub.encrypt(msg, None)[0]
```

He comments, "We can ignore this None and only need the first value from the tuple. Both exist only for compatibility." Bob is right about that and we now have a convenient textbook RSA encryption function at hand.

Run 1: RSA-Encrypted textbook DH in one line of python

Now Alice and Bob are ready for their DH exchange. In contrast to their original sketch, they will encrypt their DH values with RSA. Alice generates:

```
>>> xa = int.from_bytes(os.urandom(192), byteorder='big')
>>> ya = pow(g, xa, p)
```

and sends

```
>>> RSA_enc(k_Bob_pub, (ya, g, p))
```

Alice sends 67507dee555403ad... [504 bytes omitted]. How does Alice send the message? She hands it over to Carol. Carol starts fiddling around with with the data. "What are you doing?" Bob asks. Alice replies, "It is encrypted, those were your words. Carol will deliver the message to you."

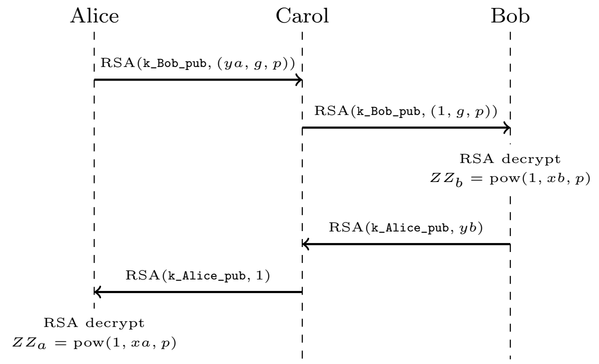
Carol forwards 23159f4e2daf11a6... [504 bytes omitted]. Bob decrypts with his private RSA key, parses ya, g, p from the message, and computes

```
>>> xb = int.from_bytes(os.urandom(192), byteorder='big')
>>> yb = pow(g, xb, p)
>>> ZZ_b = pow(ya, xb, p)
```

and sends

```
>>> RSA_enc(k_Alice_pub, yb)
```

Bob sends 86dcf718bad3ee88... [504 bytes omitted]. Carol forwards a different message. Alice performs her part to finish the DH handshake. Carol exclaims, "The key is 1!" Bob and Alice check. Carol is right. How can Carol know the established keys? Bob is right about one thing, the DH values were encrypted, so a trivial textbook DH MitM attack does not work since Carol cannot get the ya and yb values. But she doesn't need to. This is what happened so far:


Alice
Carol
Bob
RSA(k_Bob_pub, (ya, g, p))
RSA(k_Bob_pub, (1, g, p))
RSA decrypt
$ZZ_b = \text{pow}(1, xb, p)$
RSA(k_Alice_pub, yb)
RSA(k_Alice_pub, 1)
RSA decrypt
$ZZ_a = \text{pow}(1, xa, p)$

The prime p, the generator g, and the public keys are public knowledge, also known to Carol (check your textbook, neighbor). Consequently, Carol can encrypt DH values, but she cannot read the ones from Alice and Bob. Bob computes the shared DH key as $ya^{xb} \bmod p$, where Carol supplied 1 for ya. Carol can be sure that Bob will compute a shared key of 1, she doesn't need to know any encrypted values. Same goes for the exchange with Alice.

"No No," Bob protests, "these values are not allowed in DH." Alice checks RFC 2631 and quotes: «The following algorithm MAY be used to validate a received public key y [...] Verify that y lies within the interval [2,p-1]. If it does not, the key is invalid.» Bob replies, "So y = 1 is clearly invalid, you must not do this Carol." Alice objects, "The check is optional, see this all-caps MAY there?" But Bob feels certain that he is right and insists, "Any library would reject this key!"

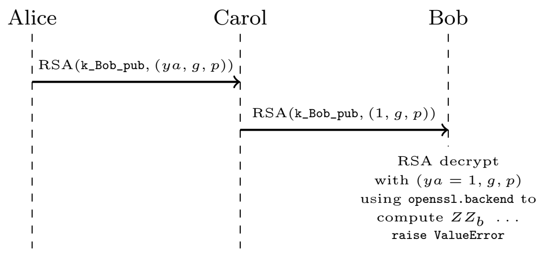
Run 2: RSA-Encrypted textbook DH using parts of the OpenSSL library

"Sure, we'll give it a try." Alice responds. She sticks to her old code because the RFC clearly states the check optional, but Bob can reject the weak values.

Alice sends 9bbc45d463d85250... [504 bytes omitted]. Carol, testing the same trick again, forwards 23159f4e2daf11a6... [504 bytes omitted]. Bob now uses `pyca/cryptography` with the openssl backend to do the DH computation. Maybe just doing `ZZ_b = pow(ya, xb, p)` was too simple? Let's see what happens when we use some part of the OpenSSL library (wrapped by `pyca/cryptography`) to perform the same computation. A word of clarification: The OpenSSL library is only used to implement the DH part on Bob's side, the exchange is not tunneled over TLS. The RSA-part remains unchanged.

```
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.backends import openssl
>>> pn = dh.DHParameterNumbers(p, g)
>>> parameters = pn.parameters(openssl.backend)
>>> xb = parameters.generate_private_key()
>>> # feed ya to the openssl library backend
>>> alice_public_key = dh.DHPublicNumbers(ya, pn).public_key(openssl.backend)
>>> assert alice_public_key.key_size == 1536 # 1536-bit MODP
group of our prime
>>> yb = xb.public_key().public_numbers().y
>>> ZZ_b = xb.exchange(alice_public_key)
```

And indeed, the last line aborts with the exception 'ValueError: Public key value is invalid for this exchange.' Alice and Bob abort the handshake. This is what happened so far:


Alice
Carol
Bob
RSA(k_Bob_pub, (ya, g, p))
RSA(k_Bob_pub, (1, g, p))
RSA decrypt with (ya = 1, g, p) using openssl.backend to compute ZZ_b ... raise ValueError

"Now you must behave, Carol. We will no longer accept your MitMed values. Now that we prohibit the two bad DH values and everything is encrypted, we are 100

Run 3: RSA-Encrypted textbook DH using parts of the OpenSSL library and custom Primes

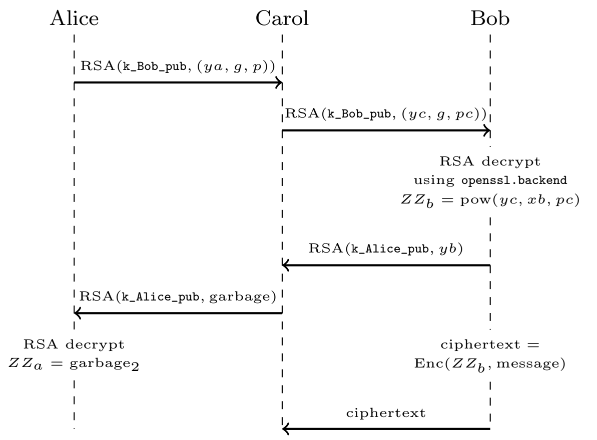
Alice and Bob try the handshake again. Carol cannot send $ya = 1$ because Bob will detect it and abort the handshake. Alice sends 09a4b88232b16136... [504 bytes omitted]. But Carol knows the math. She chooses a specially-crafted 'prime' pc and computes a random, valid yc value.

```
>>> pc = pow(2, 1536) - 1
>>> xc = int.from_bytes(os.urandom(192), byteorder='big')
>>> yc = pow(g, xc, pc)
```

Well, pc isn't actually a prime. Let's see if OpenSSL accepts it as prime. Reliably testing for primality is expensive,[37] chances are good that the prime gets waved through. Carol forwards 2f5bed0189fac5f0... [504 bytes omitted]. After RSA decryption, Bob's code with the OpenSSL backend happily accepts all values. Bob sends a790fd65fb6c163e... [504 bytes omitted]. Alice still thinks that the RFC 3526 prime is used. Carol just forwards random plausible values to Alice, but she won't be able to MitM this key. Carol forwards a7cd7cf2c5065833... [504 bytes omitted]. The DH key exchange is completed successfully. Now Bob can use the key ZZ_b established with DH to send an encrypted message to Alice.

```
>>> iv = os.urandom(16)
>>> aeskey = kdf128(ZZ_b) # squash the key to 128 bit
>>> ct = aes128_ctr(iv, aeskey, b'Hey Alice!  See, this is
perfectly secure now.')
>>> wire = ",".format(hexlify(iv).decode('ascii'), hexlify(ct)
.decode('ascii'))
```

Bob sends the IV and the ciphertext message 1f f0 07 7f f9 9a a1 19 9b bc cc c3 3d db b5 52 28 84 4f f8 8d d0 03 38 8d d6 68 81 17 73 39, ed dc cd dd d5 5f f0 0e ed d0 03 3b b8 89 9b bb b6 6a a8 8e ec c7 78 8a a0 0b b7 79 9d d3 33 32 22 27 7e ed de e9 9e ed de e6 67 7d d1 12 29 94 44 49 96 6f f5 58 8d df fe e4 4c c6 62 2c cd dd d5 52 24 4d d7 79 91 17 7e e5 5e e8 89 9e e3 32 2f f6 6e e6 6e e6 62 26 65. In summary, this is what happened so far:


Alice
Carol
Bob
RSA(k_Bob_pub, (ya, g, p))
RSA(k_Bob_pub, (yc, g, pc))
RSA decrypt using openssl.backend ZZ_b = pow(yc, xb, pc)
RSA(k_Alice_pub, yb)
RSA(k_Alice_pub, garbage)
RSA decrypt ZZ_a = garbage_2
ciphertext = Enc(ZZ_b, message)
ciphertext

Carol chose a great "prime" $pc = 2^{1536} - 1$ and knows the key is broken: Only one bit is set! She can just brute force all possible keys, the one that decrypts the ciphertext to printable ASCII text is most likely the correct key.

```
>>> iv, ct = map(unhexlify, wire.split(','))
>>> for i in range(1536):
...     keyguess = pow(2, i)
...     msg = aes128_ctr(iv, kdf128(keyguess.to_bytes(192,
byteorder='big')), ct)
...     try:
...         if not all(c in string.printable for c in
msg.decode('ascii')):
...             continue
...     except UnicodeDecodeError:  #not ASCII
...         continue
...     break
```

[37]Common primality tests are probabilistic and relatively fast, but can err. Deterministic primality tests in polynomial time exist. Note that DH does not need an arbitrary prime and some g, but the generator should generate a not-too-small$^{\text{TM}}$ subgroup.

The brute-forced key is 79,792,922,228,281,816,162,625,251,514,142,426,264,643,433,337,375,759,593,935,354,543,439,395,950,503,033,336, or in hex \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00 (exactly one bit set). Carol is correct. She immediately shouts out the message "Hey Alice! See, this is perfectly secure now." Bob is depressed. "Why doesn't my code work?", he asks. "Probably DH is not strong enough and we need to use elliptic curve DH?", he conjectures. "Maybe Carol even has a quantum computer hidden in her pocket, let me find a post-quantum replacement for Diffie-Hellman, ..." he continues. Carol interferes, "The same ideas of my attack also apply to ECDH or a post-quantum drop-in replacement with the same properties. Don't waste your time on this line of thought. If you cannot use textbook DH, ECDH (or the post-quantum candidates) won't help."

Run 4: Textbook DH signed with textbook RSA

Alice tries to put Bob on the right track, "Maybe RSA encryption does not help, but can we use RSA differently? Remember, encryption itself does not not provide integrity." "Of course," Bob replies, "we need to sign the DH values. And signing with RSA is just encryption with the private key." "Don't forget the padding," Alice is trying to help, but Bob immediately codes:

```
>>> import Crypto.PublicKey.RSA
>>> def RSA_sign(k_priv, msg):
...     # ignore the compatibility parameters
...     return k_priv.sign(msg, None)[0]
>>> def RSA_verify(k_pub, msg, signature):
...     # ignore the compatibility parameters
...     return k_pub.verify(msg, (signature, None))
```

Again, Bob is right about ignoring the compatibility parameters. However, Carol smiles as Bob completely ignored Alice's comment about padding.

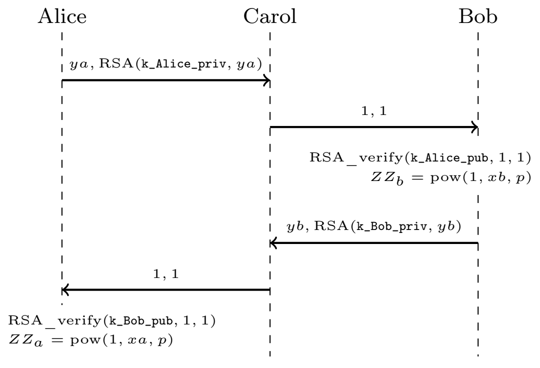
"Let's hardcode the prime p and generator g for simplicity and switch back to the trivial non-OpenSSL implementation." Alice suggests and everybody agrees. This simplifies the DH exchange as now, only y and the signature of y will be exchanged. Alice only sends the following in the first step:

```
>>> ",".format(ya, RSA_sign(k_Alice_priv, ya))
```

Alice sends 45e59717fd2ad3aa...[184 bytes of y omitted],5ee95099ea63afc6...[504 bytes of signature omitted]. Carol just forwards 1,1. Bob parses the values, verifies the signature correctly and performs his step of the DH exchange.

```
>>> ya, signature = map(int, wire.split(','))
>>> if not RSA_verify(k_Alice_pub, ya, signature):
>>>  print("Signature verification failed")
>>>  return 'reject'
[...]
>>> return ",".format(yb, RSA_sign(k_Bob_priv, yb))
```

Bob sends f543932fd7646f7e...[184 bytes of y omitted],8a3c8e3aac04e59d...[504 bytes of signature omitted]. Carol just forwards 1,1. Alice smiles as she receives the values. Nevertheless, she performs the signature verification professionally. Both the signature check at Bob and the signature check at Alice were successful and Alice and Bob agreed on a shared key. This is what happened so far, where RSA corresponds to RSA_sign as defined above:


Alice
Carol
Bob
ya, RSA(k_Alice_priv, ya)
1, 1
RSA_verify(k_Alice_pub, 1, 1)
ZZ_b = pow(1, xb, p)
yb, RSA(k_Bob_priv, yb)
1, 1
RSA_verify(k_Bob_pub, 1, 1)
ZZ_a = pow(1, xa, p)

Carol exclaims "The key is 1!" Bob is all lost, "How could this happen again? I checked the signature!" "Indeed," Carol explains, "but you should have listened to Alice's remark about the padding. RSA signatures are not just the textbook RSA operation with the private key. Plain textbook RSA is just msg^d mod N, where d is private. Guess how I could forge a valid RSA private key operation without knowledge of d if I may choose msg freely?" Bob looks desperate. "Can Carol break RSA? What is the magic math behind her attack?", he wonders. Carol helps, "1^d mod $N = 1$, for any d. Of course I did not break RSA. The way you tried to use RSA as a signature scheme is just not existentially unforgeable. Paddings, or signature schemes, exist for a reason." By the way, the RSA encryption without padding used in the previous runs is also dangerous.[38]

Run 5: Textbook DH signed with RSASSA-PSS

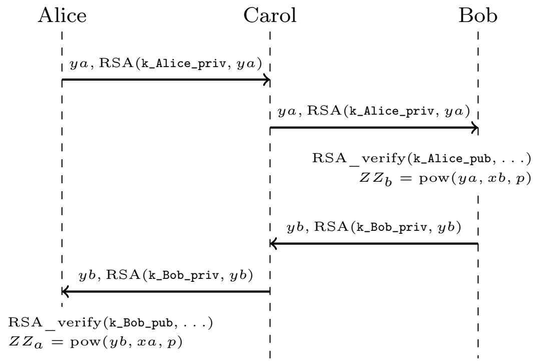
Bob replaces the sign and verify functions:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import
padding
>>> def RSA_sign(k_priv, msg):
>>>  return k_priv.sign(
...         msg,
...         padding.PSS(
...             mgf=padding.MGF1(hashes.SHA256()),
...             salt_length=padding.PSS.MAX_LENGTH
...         ),
...         hashes.SHA256()
...     )
```

The `RSA_verify` function is replaced accordingly.

Now Alice and Bob can try their handshake again. Alice sends 9403c79416ebcedb...[184 bytes of y omitted],2043516ccf286cb4...[504 bytes of signature omitted]. Carol forwards the message unmodified. Bob looks at Carol suspiciously. "I cannot modify this without breaking the signature," Carol replies. "Probably the DH prime is a bit too small for the future; Logjam predicts 1024-bit breakage. Maybe you could use fresh DH values for each exchange or switch to ECDH to be ready for the future, ... But I'm out of ideas for attack I could carry out on my slow laptop against your handshake for now." Carol concludes.

Bob sends c02a4deacd839b93...[184 bytes of y omitted],642f187cf7ca041b...[504 bytes of signature omitted]. Carol forwards the message unmodified. Finally, Alice and Bob established a shared key and Carol does not know it.


Alice
Carol
Bob
ya, RSA(k_Alice_priv, ya)
ya, RSA(k_Alice_priv, ya)
RSA_verify(k_Alice_pub, . . .)
ZZ_b = pow(ya, xb, p)
yb, RSA(k_Bob_priv, yb)
yb, RSA(k_Bob_priv, yb)
RSA_verify(k_Bob_pub, . . .)
ZZ_a = pow(yb, xa, p)

To complete the scenario, Bob uses the freshly established key to send an encrypted message to Alice.

```
>>> iv = os.urandom(16)
>>> aeskey = kdf128(ZZ_b) # squash the key to 128 bit
>>> ct = aes128_ctr(iv, aeskey, b'Hey Alice!  See, this is
perfectly secure now.')
>>> wire = ",".format(hexlify(iv).decode('ascii'), hexlify(ct)
.decode('ascii')
```

Bob sends the IV and the ciphertext message 6e e1 1c c4 48 8a ad da ad d9 97 77 7c c8 86 6a aa a4 4e e0 0b b3 38 86 65 5f fc c9 99 90 0e, 3a a4 48 82 2f f5 5f fb b0 0b b7 7d d8 83 36 6a a8 8c c0 02 21 1f fc c7 75 59 91 1e e6 67 77 7f f4 48 83 38 86 6e ec cd d8 8c c3 31 1a ab bc c3 3d d5 5e e2 25 52 21 13 3e e3 34 4c c4 4d da a5 59 94 48 89 99 96 62 29 9a a2 26 66 60 01 1c cf fc cf fc c4 4e ed d4 45 51. Carol remembers the plaintext Bob sent in run 3. She realizes that this run's ciphertext has exactly the same length as the plaintext in run 3. Carol forwards a ciphertext which is slightly shorter: 6e e1 1c c4 48 8a ad da ad d9 97 77 7c c8 86 6a aa a4 4e e0 0b b3 38 86 65 5f fc c9 99 90 0e, 37 74 43 33 35 50 0d d8 88 8a ab bc c5 53 3c ca a2 28 8f f2 21 1c c6 66 63 3d d4 4a a4 43 38 8f f4 4c cb ba a6 6f f1 18 8c cc cf f0 0e ee ee e2 24 44 4f f2 2e e6 69. Alice reads out loud the message she received and decrypted: "Encryption is not Integrity." Bob shouts, "This is not the message! How can this happen? Did Carol break AES-CTR?" Alice and Carol answer simultaneously, "AES-CTR is secure encryption, but Encryption is not Integrity."

[38]Use OAEP!