C&O 487/687: Assignment #5 Solutions

Due date: December 2, 2024 (Monday)

As the deadline for this assignment has already been extended for all students, no further extensions are available for any students. If you are requesting an accommodation because of a verification of illness form, self-declared student absence, or AccessAbility Services accommodation, your only option is to shift the weight to the final exam; see the "Missed assignment / extension policy and forms" instructions on the CO 487/687 LEARN site.

1. [3 marks] **Digital signature schemes.**

   In the lecture slides, you have been provided with the definition of *existential unforgeability under chosen message attack*. In this definition, an adversary has access to a signing oracle, which, when queried on message m, will produce a signature s such that (m, s) is a valid message/signature pair. The adversary's goal is to come up with a valid message/signature pair (m^*, s^*), where m^* is not a message that they previously queried to the signing oracle.

   Sometimes, we care about a different, stronger version of unforgeability called **strong** existential unforgeability under chosen message at tack. In this definition, the adversary still has access to a signing oracle, which, when queried on message m, will produce a signature s such that (m, s) is a valid message/signature pair. However, this time the adversary's goal is to come up with a valid message/signature pair (m^*, s^*), where s^* was never the result of an oracle query on m^*.

   In other words, in the strong unforgeability definition, the adversary **is** allowed to query m^* to the oracle, but they cannot use the output s^* of such a query as their forged signature. To say it another way, there are two ways for the adversary to win: (1) the adversary produces an entirely new message and valid signature (m^*, s^*) for some m^* that wasn't queried, or (2) the adversary produces a new valid signature s^* on a message m^* that was previously queried; in case (2), the signature s^* must be different from all the signatures that the adversary received in response to their signing queries for m^*.

   Show that ECDSA does not satisfy strong existential unforgeability.

   Solution. Suppose an attacker queried the oracle on message m and received back the signature (r, s). Then $(r, -s)$ is also a valid signature for m, since

$$\begin{aligned} x\left(\frac{H(m)}{-s}P + \frac{r}{-s}(\alpha P)\right) &\equiv x\left(\frac{1}{-s}(H(m) + r\alpha)P\right) \bmod q \\ &\equiv x\left(\frac{1}{-s}(ks)P\right) \bmod q \\ &\equiv x(-kP) \bmod q \end{aligned}$$

   where $x(Q)$ denotes the x-coordinate of a point Q on the elliptic curve. Since elliptic curve points form an additive group with identity $\mathcal{O}$, we have that $kP + (-kP) = \mathcal{O}$, and hence that $x(kP) = x(-kP)$ (and $y(kP) = -y(-kP)$, where $y(Q)$ is the y-coordinate of a point Q on the elliptic curve). So, continuing our algebra from above, we get that

$$x(-kP) \equiv x(kP) \equiv r \bmod q$$

   as desired.

2. [13 marks] **Public key infrastructure.**

   After several months of being thwarted by clever CO 487 students, the engineers are making their last ditch attempts to undermine mathematics students. The Engineering Committee for Disrupting Student Assessment (ECDSA)[1] has been trying to impersonate TAs and make fake posts on Piazza to trick math students into believing that their final exams have been cancelled.

[1] The Engineering Committee for Developing Silly Acronyms has also been very busy this term.

So far, I have managed to delete all their fake posts before anyone saw them, but to prevent this from happening in the future, I am considering having all announcements on Piazza be signed using elliptic curve digital signatures. To distribute the TA's public keys, I would set up a public key infrastructure to issue certificates containing the TA's public key and the certificates would be signed by me (`dstebila`).

In `a5pki.zip` (available on LEARN), you'll find most of the code for this system, as well as some public keys, certificates, and messages. These scripts again use the Python cryptography library that we previously used in assignments 3 and 4. (Follow the instructions from assignment 3 in order to get Python up and running for this question or use the UW Jupyter server.)

This ZIP file contains the following files:

- `key_gen.py`: A Python file for generating a user's public key and secret key.
- `dstebila.pk`: The public key of the root certificate authority.
- `generate_certificate.py` A Python file for generating a user's certificate by using the CA's signing key to sign the user's public key and other data fields.
- `*.cert`: The certificates of the TAs, containing their public keys.
- `sign.py`: A Python file for generating signatures on messages.
- `message*.signed.txt`: Some alleged messages from the TAs with information about the final exam.
- `verify.py` or `verify.ipnyb`: The skeleton for a Python file for verifying signed messages, with some bits missing for you to fill in. (These two files contain the same contents, and you can use either of them, depending on whether you want to use Python or Jupyter.) This will be the only file you need to edit.

Your task is to finish writing `verify.py` or `verify.ipynb` and determine which messages are legitimate.

(a) [5 marks] Describe (in words or in pseudocode) the things you need to check in order to see if you should trust the signature on a message.

Solution.

- Was the certificate issued by the trusted CA or someone else?
- Does the CA's signature on the certificate verify?
- Is the current date within the certificate's prescribed validity period (not too early, not expired)?
- Does the signer's signature on the message verify?
- Is the certificate's key usage for digital signatures?

(b) [2 marks] Finish writing `verify.py` (if you are working on your own computer) or `verify.ipynb` (if you are working on UW's Jupyter server at `https://jupyter.math.uwaterloo.ca`).

If you are working on your own computer, make sure all the files are in the same directory and then run `python3 verify.py`

If you are working on UW's Jupyter server `https://jupyter.math.uwaterloo.ca`, upload all the files from the ZIP file into the same directory. Make sure you're using the "Python 3 extras" kernel for the Jupyter notebook.

Submit the source code for your script as a PDF or screenshot to Crowdmark.

Solution.

```
from datetime import datetime
import json
import sys
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes
```

```
from cryptography.hazmat.primitives.asymmetric.utils import encode_dss_signature
from cryptography.hazmat.primitives import serialization

# Helper function for converting strings into byte arrays needed by cryptographic functions
def string_to_bytes(s):
    return s.encode('utf-8')

# This function will ensure that we represent the JSON dictionary as exactly the
# same string every time, otherwise we'd get different hashes while signing
def canonicalize_json(j):
    return json.dumps(j, sort_keys=True)

def verify(ca_identity, signed_message_filename):

    print("Trying␣to␣verify␣" + signed_message_filename)

    # Load the signed message data
    with open(signed_message_filename, 'r') as fh:
        signed_message = json.load(fh)

    # Read out the identity of the signer and load their certificate
    signer_identity = signed_message['signer␣identity']
    with open(signer_identity + '.cert', 'r') as fh:
        signer_cert = json.load(fh)
    # Format the certificate body for signing as a byte array in a canonical order
    cert_body_to_be_signed = string_to_bytes(canonicalize_json(signer_cert["body"]))

    # Read out the identity of the issuer and load their public key
    issuer_identity = signer_cert['body']['issuer␣identity']
    signer_pk = serialization.load_pem_public_key(string_to_bytes(signer_cert['body']['public␣key']))
    with open(ca_identity + '.pk', 'r') as fh:
        ca_public_key = serialization.load_pem_public_key(string_to_bytes(fh.read()))

    # YOUR SOLUTION STARTS HERE

    # Check that issuer of the certificate is the CA we trust
    if issuer_identity != ca_identity:
        print("The␣certificate␣used␣was␣signed␣by␣" + issuer_identity
            + ",␣not␣the␣trusted␣CA␣(" + ca_identity + ")")
        return False

    # Check that the current date and time is within the validity period of the certificate
    if not(datetime.fromisoformat(signer_cert['body']['validity␣start'])
        <= datetime.now()
        <= datetime.fromisoformat(signer_cert['body']['validity␣end'])):
        print("The␣certificate␣for␣" + signer_identity
            + "␣is␣not␣currently␣valid␣(either␣too␣soon␣or␣expired).")
        return False

    # Check the CA's signature on the certificate
    cert_sig = encode_dss_signature(signer_cert["signature"]['r'],
                                    signer_cert["signature"]['s'])
    try:
        ca_public_key.verify(cert_sig, cert_body_to_be_signed, ec.ECDSA(hashes.SHA256()))
    except:
        print("The␣CA's␣signature␣on␣" + signer_identity + "'s␣certificate␣is␣invalid")
        return False

    # Check the signer's signature on message
    message_sig = encode_dss_signature(signed_message["signature"]['r'],
                                       signed_message["signature"]['s'])
    try:
        signer_pk.verify(message_sig, string_to_bytes(signed_message['message']),
                         ec.ECDSA(hashes.SHA256()))
    except:
        print("The␣signature␣by␣" + signer_identity + "␣on␣the␣message␣is␣invalid")
        return False

    # Check that the key usage purpose is for signing, not public key encryption
    try:
        assert signer_cert["body"]["key␣usage"] == "digital␣signature"
    except:
        print(f"The␣key␣usage␣type␣for␣{signer_identity}␣is␣not␣'digital␣signature',")
        print(f"it's␣'{signer_cert["body"]["key usage"]}'")
        return False

    print("The␣following␣message␣was␣verified␣as␣being␣signed␣by␣" + signer_identity + ":")
    print(signed_message['message'])
    return True
```

```
verify("dstebila", "message1.signed.txt")
verify("dstebila", "message2.signed.txt")
verify("dstebila", "message3.signed.txt")
verify("dstebila", "message4.signed.txt")
verify("dstebila", "message5.signed.txt")
verify("dstebila", "message6.signed.txt")
```

(c) [6 marks] Which of the 6 signed messages in the ZIP file can you verify as legitimate? For the messages that are not legitimate, what is the reason?

Solution.

- `message1`: The certificate from `youcef` is not valid (on closer inspection, it is self-signed!).
- `message2`: The certificate from `josephine` is fine, but the signature on the message is not valid.
- `message3`: The CA's signature on the certificate for `camryn` is not valid.
- `message4`: The certificate from `nic` is for the future and thus not valid.
- `message5`: The certificate and message from `fernanda` are valid.
- `message6`: The certificate for `santiago` is to be used for `public key encryption`, not digital signatures.

3. [6 marks] **One-time passwords.**

In 2020, the University of Waterloo made two-factor authentication mandatory, to increase security against weak/reused passwords, password database breaches, and phishing attacks. UW's two-factor system uses a commercial product called Duo, which supports several authentication methods: (i) a push notification to a proprietary mobile app with a confirmation code; (ii) a telephone call or SMS message; (iii) a one-time passcode (needed for VPN access from off campus); (iv) WebAuthn Passkeys (`https://en.wikipedia.org/wiki/WebAuthn`).

The third option, one-time passcodes, is based on a standardized protocol called HOTP (HMAC-Based One-Time Password, `https://tools.ietf.org/html/rfc4226`, see also Wikipedia).

HOTP works as follows to generate a 6-digit one-time password.

When a user initially registers with a server, the server assigns the user a random secret 128-bit key k, which the server stores in its database, and which the user embeds in their HOTP authenticator application. The user and the server initialize a counter c to 0.

Each time the user wants to log in to a server, they press the "generate OTP button" in their application which does the following operations:

(1) Compute $x \leftarrow \text{HMAC-SHA256}(k, c)$.

(2) Let i be the last 4 bits of x, represented as a non-negative integer.

(3) Let y be bits i through $i+30$ inclusive of x (i.e., y is 31 bits long), represented as a non-negative integer.

(4) Compute $z \leftarrow y \bmod 10^6$.

(5) Return z as the one-time password.

(6) Increment counter c and save the new counter value.

When the server receives a login request, it looks up the k and c for the user in question and does the same operations as above, then compares what it has computed against what the user sent. If the values match, the server grants the user access, and the server increments the counter c and saves the new counter value. If the values do not match, the server rejects the login request and does not update the counter c. Many servers also impose rate-limiting on login requests, for example, locking the account after more than 10 failed login attempts.

(a) [2 marks] Suppose an attacker has compromised your password, and observed several past OTPs you have used, but does not have your OTP key k. Describe the best attack (with feasible runtime) you can think of to remotely break into the account. What is the probability of success of your attack? Assume that HMAC-SHA256 is a secure pseudorandom function.

Solution. Since our runtime must be feasible, we cannot do a brute force search for the key k. Since HMAC-SHA256 is a secure pseudorandom function, its output, for an unknown k but even a known c, will still look random; and knowing past outputs does not help predict future outputs. Hence the next z we need to log in is a uniform 6-digit decimal number. If we have 10 login attempts, then our best attack is just to guess 10 different random 6-digit numbers, which has a success probability $10/100000 = 1/10000$.

(b) [2 marks] Suppose that the counter in HOTP was not updated by either party, so that each time the user wants to log in to a server it computes $x \leftarrow \text{HMAC-SHA256}(k, 0)$. Under what attack conditions does this change make it easier for an attacker to gain access to your account? Would UW's system be vulnerable if this was used?

Solution. Suppose an attacker can observe the z values sent by the user. If there is no counter, then, because HMAC-SHA256 is deterministic, the x, i, and z values will be the same every time for the same k. Thus the user always uses the same "one-time" password, completely undermining the "one-time"-ness. In order for an attacker to make use of this, the attacker would have to be able to see your one-time password when it's submitted. Since UW's log-in system is over HTTPS, this is prevented. However, if an attacker carries out a phishing attack that convinces a user to enter the OTP on an attacker-controlled website, then the attacker has everything they need to impersonate the user.

(c) [2 marks] An alternative to HOTP is TOTP (Time-based One-Time Password, `https://tools.ietf.org/html/rfc6238`, see also Wikipedia). Briefly investigate TOTP. What is the main difference between TOTP and HOTP? Do you think one is more secure than the other? Is one easier to use than the other?

Solution. Instead of using a counter, TOTP uses the current time (rounded to 30 second increments). There isn't really a difference in security. The main benefit to TOTP is that you don't have to worry about the user's counter and the server's counter getting out of synchronization (for example, because the user pressed the "generate OTP" button multiple times without sending all those values to the server). However it does require that the clock on the client and the server be roughly synchronized.

(d) [0 marks] I encourage you to investigate Passkeys, which are based on digital signatures. Your browser creates a digital signature key pair for each site, and uploads the public key to that site at registration time. At runtime, your browser authenticates to the server by signing a challenge using the corresponding private key. Passkeys are stored in your browser or operating system's password manager, and can be synchronized across devices if your password manager supports (e.g., Google password manager, Apple iCloud keychain). On Apple devices, for example, this enables you to do your two factor authentication on University of Waterloo Duo by unlocking your passkey using your fingerprint / face scan, and these can be synchronized across all your Apple devices. Similarly for Google devices and the Chrome browser.

- https://arstechnica.com/information-technology/2023/05/passwordless-google-accounts-are-easier-and-more-secure-than-passwords-heres-why/
- https://arstechnica.com/information-technology/2023/05/passkeys-may-not-be-for-you-but-they-are-safe-and-easy-heres-why/
- https://safety.google/authentication/passkey/
- https://blog.google/technology/safety-security/the-beginning-of-the-end-of-the-password/
- https://support.apple.com/en-ca/guide/iphone/iphf538ea8d0/ios
- https://learn.microsoft.com/en-us/windows/security/identity-protection/passkeys/?tabs=windows%2Cintune

4. [7 marks] **Let's steal a Tesla.**[2]

[2] Please do not steal cars.

One feature that Tesla cars are known for is their passive keyless entry and start (PKES) system, which automatically unlock and start a user's car when the user (in possession of the paired key fob) is in close physical proximity. The key fob and car share a 2-byte car identifier id and a 40-bit secret key, k.

The system employs a (proprietary and obsolete) cipher called DST40 as a pseudorandom function. In particular, the DST40 pseudorandom function takes as input a 40-bit key and a 40-bit challenge, and produces a 24-bit response. For the purposes of this question, you can assume that DST40 produces output indistinguishable from random, other than the fact that it has very small inputs and outputs.

The PKES protocol for a Tesla Model S is as follows:

Step 1. The car periodically broadcasts (over radio frequency) its 2-byte car identifier id. The broadcast is low powered, so only entities in a given (and small) range will be able to hear these broadcasts.

Step 2. The key fob is always listening for broadcasts of car identifiers. If it hears the identifier id of the car it is paired with, it will send the car a response, letting it know that it is in range and requesting a challenge.

Step 3. The car will broadcast a 40-bit challenge, ch.

Step 4. Upon receiving ch, the key fob will send back a response, which it computes using the DST40 cipher and the shared secret key. In particular, the key fob will send back the 24-bit response $rsp = \mathsf{DST40}(k, ch)$.

Step 5. Upon receiving rsp, the car will verify that $rsp = \mathsf{DST40}(k, ch)$. If so, it will unlock and start.

(a) First, let's look at the DST40 cipher.

(i) [1 marks] For a given challenge ch and a response rsp, how many on average keys map ch to rsp? Assume that there is a uniform distribution of responses over the possible challenges and keys.

Solution. For a given challenge, there are $\frac{2^{40}}{2^{24}} = 2^{16}$ keys that map to each response.

(ii) [2 marks] How many challenge/response pairs do you need to observe to determine the key (with high probability)?

Solution. We need 2 challenge/response pairs. The first pair gives us a subset consisting of 2^{16} keys. By the birthday paradox, it takes about 2^{20} keys before we can expect to find a collision for a random challenge ch (that is, two keys, k and k' such that $\mathsf{DST40}(k, ch) = \mathsf{DST40}(k', ch)$). So, we can expect that, for a randomly chosen challenge, each of the remaining 2^{16} possible keys will give a different response.

(iii) [2 marks] Describe a key-recovery attack on the DST40 pseudorandom function that can be executed quickly. You may assume that you have the number of challenge/response pairs computed in the previous part of the question, and that you get to pick what the challenges are in each pair. You may assume that you have access to a precomputed list $L = [\mathsf{DST40}(k_i, ch), k_i, ch]$ for a fixed, public challenge ch, and all keys k_i, which is sorted by the response given. You may also assume that it takes about 2 seconds to compute 2^{16} DST40 operations on your computer.

Solution. Let the two challenge/response pairs we have access to be (ch, r) and (ch', r'), where ch is the challenge that L is based off of. Since L is sorted by response, we can quickly find the list L' of the 2^{16} keys from L which give the response r for challenge ch. Now, for each key k_i in L', compute $\mathsf{DST40}(k_i, ch')$ until we find the key which gives us the response r'. Since it only takes 2 seconds to compute 2^{16} DST40 operations, this can be done quickly.

(b) [2 marks] Describe how you can create your own key fob clone in order to steal a Tesla Model S. You may assume you have the list L from part (a), and that you have the opportunity to be in close proximity to the car and to the key fob whenever needed.

Solution. Our attack is as follows:

- First, stand near the victim car and record when it broadcasts its 2-byte identifier.
- Go near the owner of the car and broadcast the car's 2-byte identifier. When the response from the key fob is received, send the fob two challenges: ch and ch', where ch is the challenge that the precomputed list L used, and ch' is chosen at random, and receive back the responses r and r', respectively.
- Use r to compile the list L' from part (a.iii), and use r' to determine which key $k \in L'$ is correct, as in part (a).
- Once you have the key k, you can impersonate the key fob by responding to any challenge ch'' sent by the car with $\mathsf{DST40}(k, ch'')$.

This question is based on a real attack on Tesla cars, identified by researchers in 2018. You can read about this at

https://www.esat.kuleuven.be/cosic/news/fast-furious-and-insecure-passive-keyless-entry-and-start-in-modern-supercars/

and watch a video of grad students carrying out the attack at

https://www.youtube.com/watch?v=aVlYuPzmJoY

5. [8 marks] **Invalid padding attacks.**

Note: Although this question refers to the TLS protocol, that is only for motivation. The technical part of the question only relies on concepts from earlier sections of the course: block cipher modes of operation and authenticated encryption. You can do this question without the material from the TLS lecture.

Of all the squads of devious and treacherous engineering students, the Totally Legendary Squad (TLS) is the most devious and treacherous of them all. They plan on taking down the CO 487 students by attacking them at their root, by taking out their leader, Prof. Stebila. Earlier this term, they let a bear into his office, in the hopes that it would eat him, but luckily, one of his loyal TAs caught wind of the plan and equipped him with a bear bell just in time.

For a plot of this level of sophistication and audacity, a multitude of engineering students are required in order to carry it out. To communicate with so many TLS members, they use a private online forum. Since security is of utmost importance, they use their namesake algorithm, TLS 1.0, to create a secure channel over which to send messages to the forum.

In TLS v1.0, messages are encrypted by first padding them using the PKCS #7 padding scheme, and then encrypting them using a 128-bit block cipher in CBC-mode. The PKCS #7 padding scheme works as follows:

Step 1. Determine how many bytes of padding are needed. Call this number n.

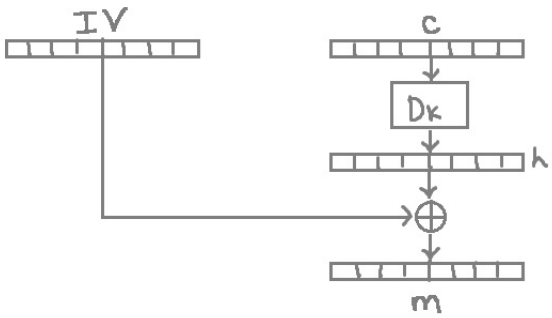
Step 2. Pad the message by appending n bytes, each of which is the number n represented as a byte.

So, if the message needs one byte of padding, it would be padded with a single byte `01` (written here in hexadecimal for convenience), if it needs two bytes of padding, it would be padded with two bytes `0202`, if it needs three bytes of padding, it would be padded with `030303`, etc.

When decryption is performed, first, CBC-decryption is performed, and then an additional check is performed to determine if the padding is valid. If the padding is not valid, an "`invalid padding error`" alert is returned to the sender in plaintext over the network.

(a) [3 marks] Suppose you intercepted a one-block ciphertext, c, which Alice sent to a TLS server. Describe how you can efficiently recover the last byte of the message. You may assume that the TLS server will automatically (attempt to) decrypt any ciphertexts sent to it, and will return an `invalid padding error` message if the padding is not valid, but will not return the plaintext if decryption is successful; and that it will keep using the same secret key throughout your attack. Justify why your attack works and why it is efficient. How many queries do you have to make?

Solution. Let c be the ciphertext block we intercepted, and let IV be the IV that was used to encrypt it (note that we also know this value as it is sent in the clear alongside the ciphertext). Recall CBC decryption for a one-block message:



In particular, notice that

$$h = m \oplus IV$$

Notice also that if we decrypt c using a different IV, then the value of m will change, but the value of h will not. Our strategy will be to solve for h, and then use this value, and the equation above, to solve for m.

To do this, let x be a new IV that we'll continually modify our attack. To start with set x to be any bitstring. If querying (x, c) does not return an invalid padding error, then keep choosing new values of x until it does (fewer than half the possible message blocks have valid padding, so with high probability this will happen on the first try). Then, iterate over all possible values for the last byte of x until querying (x, c) does not return an invalid padding error. Since we've only been changing the last byte of x, only the last byte of the corresponding message m' has been changing as a result. So, the most likely way to get valid padding by doing this is if XOR-ing h with x results in a message m' whose last byte is `01` (valid padding could also occur if the fixed second-last byte is `02` and the last byte happens to hit `02` before hitting `01`, if the fixed second- and third-last bytes are `0303` and the last byte happens to hit `03` before hitting `01`, etc, but we can check for this by changing the second last byte of x, and re-querying the padding oracle to see if it still accepts, and if it does, then we know that we are in the 01 case and can stop searching). When this happens (and it will happen after at most 256 queries), we can compute the last byte of h as

$$h_{16} = \texttt{01} \oplus x_{16}.$$

To recover the last byte of m, we can use our first equation and compute

$$m_{16} = h_{16} \oplus IV_{16}.$$

(b) [2 marks] Extend your attack to recover the entire message. (If your attack recovers all but the first byte of the message, then that is fine.) Justify why your attack extension works, and why it is efficient. How many queries do you have to make?

Solution. Now, we'll consider the last two bytes of h. In this case, the most likely way to obtain valid padding is if these bytes are `0202`. Since we already know the value of h_{16}, we can compute that for this case, the last byte of x should be

$$x_{16} = \texttt{02} \oplus h_{16}.$$

Then, just as in part (a), we can iterate over all possible values for the second last byte of x until querying (x, c) does not return an invalid padding error. Since we've only been changing

the second last byte of x, only the second last byte of the corresponding message m' has been changing as a result. So, the only way to get valid padding by doing this is if XOR-ing h with x results in a message m' whose second last byte is `02` (in this case, we have no other possibilities to consider, since we already set the last byte to be `02`). When this happens (and it will happen after at most 256 queries), we can compute the second last byte of h as

$$h_{15} = \texttt{02} \oplus x_{15}.$$

We can repeat the above steps for padding of the form `030303`, `04040404`, etc, until we have recovered (almost) all bytes of the message. Note that we can't recover the first byte, since sending an empty message is not possible.

In total, this attack takes at most 15×256 queries, since we are recovering 15 of the 16 bytes in the block.

It is worth noting that it is possible to recover the first byte of the message if we queried the padding oracle on a two-block ciphertext xc (and any random IV), where x is a ciphertext block which we control in exactly the same way as we control the IV in the attack above, and c is the ciphertext block which we are trying to decrypt. This way, if the last block is `0F0F...0F` then the message will not be empty.

(c) [1 marks] Can this attack be modified to decrypt longer messages? Give a brief, 1-2 sentence explanation of how you would go about doing this, or why this wouldn't work.

Solution. Yes, this attack can be modified to decrypt longer messages. You can apply the attack above to each ciphertext block separately in order to recover (most of) the plaintext.

(d) [1 marks] TLS 1.0 offers an optional MAC to provide message integrity. If this option is chosen, then an authenticated encryption scheme is formed using the MAC-then-pad-then-encrypt paradigm. Is this effective at preventing your attack? Justify your answer.

Solution. No, this would not prevent the attack above. Since decryption is performed before authentication, the adversary would still have access to the same padding oracle. The only difference is that the ciphertext would be a little longer, since it would also contain the encryption of the tag.

(e) [1 marks] Would a pad-then-encrypt-then-MAC option be effective at preventing your attack (assuming that the IV is MAC-ed alongside the ciphertext)? Justify your answer.

Solution. Yes, this would prevent the attack above. Assuming the MAC is secure, then we would not be able to compute valid tags for each of our chosen ciphertexts. So, the authentication step would fail on each of our queries, and decryption would never be performed by the server, and we would therefore lose access to our padding oracle.

SSL version 3 and TLS version 1.0 really were vulnerable to this attack. Fixing this was the main purpose of version 1.1 of the TLS standard. Surprisingly, it took nearly 6 years for researchers to discover this vulnerability after SSLv3 was released.

Academic integrity rules

You should make an effort to solve all the problems on your own. You are also welcome to collaborate on assignments with other students in this course. However, solutions must be written up by yourself. If you do collaborate, please acknowledge your collaborators in the write-up for each problem. *If you obtain a solution with help from a book, paper, a website, or any other source, please acknowledge your source. You are not permitted to solicit help from other online bulletin boards, chat groups, newsgroups, or solutions from previous offerings of the course.*

Due date

The assignment is due via Crowdmark by 11:59:59pm on December 2, 2024.

As the deadline for this assignment has already been extended for all students, no further extensions are available for any students. If you are requesting an accommodation because of a verification of illness form, self-declared student absence, or AccessAbility Services accommodation, your only option is to shift the weight to the final exam; see the "Missed assignment / extension policy and forms" instructions on the CO 487/687 LEARN site.