CO 487 - Assignment 5

Bilal Khan (b54khan)
bilal.khan@student.uwaterloo.ca

December 2, 2024

Contents

1 1

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 attack. 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 new message and valid signature (m^*, 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.

The signature for a message verified with ECDSA uses a randomly generated value k as part of the signature. There are no restrictions on it being derived from some IV or key, so it is possible to generate a new signature for the same message by just generating a new random k and recomputing the signature (in the unlikely event that the same signature is generated twice, keep generating k until you get a different signature).

2 2

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. 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.

2.1 a

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.

- Verify that the CA issued the certificate
- Verify the certificate's signature using the CA's public key
- Verify that the certificate's time period is valid
- Verify that the message's signatrue is valid using the cert's publick key

2.2 b

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.

```
try:
    if issuer_identity != ca_identity:
        raise Exception('issuer_identity != ca_identity')

    cert_signature = encode_dss_signature(signer_cert['signature']['r'],
                                          signer_cert['signature']['s'])

    try:
        ca_public_key.verify(cert_signature, cert_body_to_be_signed,
                             ec.ECDSA(hashes.SHA256()))
    except Exception as e:
        raise Exception('ca_public_key.verify=False')

    current_time = datetime.now()
    validity_start = datetime.fromisoformat(
        signer_cert['body']['validity start'])
    validity_end = datetime.fromisoformat(
        signer_cert['body']['validity end'])

    if current_time < validity_start or current_time > validity_end:
        raise Exception(
            'current_time < validity_start or current_time > validity_end')

    message_signature = encode_dss_signature(
        signed_message['signature']['r'], signed_message['signature']['s'])

    try:
        signer_pk.verify(message_signature,
                         string_to_bytes(signed_message['message']),
                         ec.ECDSA(hashes.SHA256()))
    except Exception as e:
        raise Exception('signer_pk.verify=False')

    print('verified')
    return True
except Exception as e:
    print('not verified: ' + str(e))
    return False
```

2.3 c

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?

```
$ python verify.py
Trying to verify message1.signed.txt
not verified: issuer_identity != ca_identity
Trying to verify message2.signed.txt
not verified: signer_pk.verify=False
Trying to verify message3.signed.txt
not verified: ca_public_key.verify=False
Trying to verify message4.signed.txt
not verified: current_time < validity_start or current_time > validity_end
Trying to verify message5.signed.txt
verified
Trying to verify message6.signed.txt
verified
```

3 3

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:

- a push notification to a proprietary mobile app with a confirmation code;
- a telephone call or SMS message;
- a one-time passcode (needed for VPN access from off campus);
- 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:

- Compute $x \leftarrow \text{HMAC-SHA256}(k, c)$.
- Let i be the last 4 bits of x, represented as a non-negative integer.
- Let y be bits i through $i+30$ inclusive of x (i.e., y is 31 bits long), represented as a non-negative integer.
- Compute $z \leftarrow y \mod 10^6$.
- Return z as the one-time password.
- 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.

3.1 a

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.

The OTP has 6 characters so random guessing across ten attempts, you have a 0.001% chance of success. Rejecting any possible guesses that have already been used in past OTPs should improve this probability.

3.2 b

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?

If the same counter is used for all OTPs, then under the condition that the attacker has compromsed your password and has access to at least one past OTP, they can reuse the same OTP to login.

3.3 c

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?

TOTP uses the current time (chunked into a certain interval e.g. 30s) as the coutner. This is more secure because its harded to brute force the OTP, the counter changes every 30s so the attacker has to restart all brute force attempts from scratch every 30s instead of every time there is a successful login (HTOP). HOTRP is easier to use since you don't have a time limit to quickly cpy the code and enter it.

3.4 d

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.

N/a

4 4

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:

- 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.
- 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.
- The car will broadcast a 40-bit challenge, ch.
- 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 = DST40(k, ch).
- Upon receiving rsp, the car will verify that rsp = DST40(k, ch). If so, it will unlock and start.

4.1 a

First, let's look at the DST40 cipher.

- 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.

  2^{40} keys match to 2^{24} responses so $2^{40}/2^{24} = 2^{16}$ keys map ch to rsp assuming uniform dist.

- How many challenge/response pairs do you need to observe to determine the key (with high probability)?

  There are 2^{40} possible keys. Each challenge/response pair is independent and reduces the search space by 2^{24} since each 40-bit challenge maps to a 24-bit response. With one pair the expected number of keys you need to search to get a hit is 2^{16}. With two pairs the expected number of keys is $2^{16}/2^{24} = 2^{16-24} = 2^{-8} = 1/256$, or that the probability of a wrong key satisfying the response is 1/256, so two pairs is enough to find the key in most cases.

- 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 = [DST40(ki, ch), ki, ch] for a fixed, public challenge ch, and all keys ki, which is sorted by the response. You may also assume that it takes about 2 seconds to compute 216 DST40 operations on your computer.

For the public challenge, get the corresponding response. Use the precomputed list to find all 2^{16} keys that correspond to that response (neglible time, can binary search since the list is already sorted by response). For all 2^{16} key candidates, compute the responses for some random new challenge using $DST40$. For this challenge, get the corresponding true response as well. The key who's response matches the true response is the correct one with probability 255/256. This takes just over 2 seconds.

4.2 b

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.

Capture the car id and use it to spoof the car. Send the key fob the two challenges and get the responses. Use the list to find the key, then spoof the key fob: listen for the car id, spoof the response, recieve the challenge from the car, use the key to compute the response to the challenge $DST40(k, ch)$, send it to the car which will unlock.

5 5

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

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

5.1 a

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?

Since this is a single-block message, CBC decrypttion works as $m = D_k(c) \oplus IV$. If we decrypt a modified ciphertext c', we get $m' = D_k(c') \oplus IV$. Looking only at the last byte of the message: $m_{16} = D_k(c)_{16} \oplus IV_{16}$ and $m'_{16} = D_k(c')_{16} \oplus IV_{16}$.

If we replace the last byte of c with a byte b to get c', the difference between the decrypted bytes ($D_k(c)$ and $D_k(c')$) is $c_{16} \oplus b$, so we have $D_k(c')_{16} = D_k(c) \oplus c_{16} \oplus b$.

$$
\begin{aligned}
m_{16} &= D_k(c)_{16} \oplus IV_{16} \\
m'_{16} &= D_k(c')_{16} \oplus IV_{16} \\
&= (D_k(c) \oplus c_{16} \oplus b) \oplus IV_{16} \\
m'_{16} &= m_{16} \oplus c_{16} \oplus b \\
m_{16} &= m'_{16} \oplus c_{16} \oplus b
\end{aligned}
$$

This will let us recover (m_{16}) given that we know (m'_{16}), which was created by replacing c_{16} with b_{16}.

We know that the server will return an error if the padding of the decrypted message is invalid. We also know that one valid decrypted padded message is one where the last byte is 01. So, now we need to try all 256 possible values of b and send the modified ciphertext to the server. The only value of b that will not return an error is the one that makes $m'_{16} = 01$. We can assume the

After we have this value of b. we can compute $m_{16} = 01 \oplus c_{16} \oplus b$. This requries 256 queries.

5.2 b

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?

We can extend this attack to target the i-th byte of the message. For all bytes after the i-th byte, xor them with whatever byte value will result in that byte being a valid padding byte for a message with i message bytes and $16 - i$ padding bytes (e.g. 01 for $i = 15$, 0202 for $i = 14$, ...). We can then send all possible values for the i-th byte to the sever to see which results in a valid padding and use the same $m_{16} = 01 \oplus c_{16} \oplus b$ method to recover the i-th byte, but replacing 01 with the corresponding valid padding byte. For a 16-byte / 128-bit message this requires $16 * 256 = 4096$ queries.

5.3 c

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.

Yes, each block can have some padding bytes and we can attack each byte of each block individually.

5.4 d

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.

No, paddign verification happens after the mac in the encrypting and before mac verification in the decryption, so it won't stop it unless the error handling behaviour changes to not short circuit.

5.5 e

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.

Yes, putting the mac at the end and short circuiting error handling will detect the modified ciphertexts.