# עבודת הגשה מספר 2

קורס: נושאים מתקדמים בפרטיות ואבטחת תוכנה (67515)

מגיש: דור מסיקה, ת.ז: 318391877

31 בינואר 2023

## ארכיטקטורה

**ה־$PKI$ שבניתי מורכב מ־4 מחלקות:**

$CA$ - מחלקה עבור אובייקטים המייצגים ישויות המורשות לבצע חתימות על גופים אחרים. המחלקה מכילה בין היותר תכונות של מפתח פומבי ופרטי, תעודה אישית ורשימת תעודות שבוטלו ($Revoked\ List$). בנוסף, המחלקה כוללת מתודות למתן תעודות לגופים אחרים והוספת תעודה לרשימת אלו שבוטלו. נציין שבכל הוספה של תעודה חדשה ל־$Revoked$, תתבצע בדיקה עבור כל אלו הקיימות במערך וימחקו אלו שכבר פגות תוקף מבחינת ה־$Validity\ date$ שלהן. המחלקה יורשת מ־$Entity$ ובכך מקבלת גם באופן בסיסי את הכוח לחתום על אובייקטים.

$Entity$ - מחלקה עבור גופים המורשים לחתום על אובייקטים. המחלקה מכילה תכונות של מפתח פומבי, מפתח פרטי ותעודה אישית, ומתודה המקבלת גוף ומחזירה חתימה עבורו. בנוסף, המחלקה מכילה גם מתודת $set$ להשמת תעודה חדשה ו־$get$ לקבלת התעודה הקיימת.

$Relying\ Party$ - מחלקה עבור ביצוע פעולת ה־$Verify$, אשר צריכה לקבל מראש את רשימת ה־$RootCAs$ ומילון הממפה שם של $Entity/CA$ לרשימת ה־$Revoked$ שלו, כך במקום העובדה שבמציאות אנו יכולים לחפש רשימות אלו באינטרנט. בנוסף, היא מקבלת בכל הרצה של פונקציית ה־$Validation$ את האובייקט הנבדק ואת התעודה "הנמוכה ביותר" בשרשרת התעודות (של הגוף שחתם על האובייקט).

$Object$ - מחלקה המייצגת אובייקט אשר דרושה חתימה עבורו. מכיל שדה של ערך ושל חתימה, ומתודת $set$ לעידכון החתימה שלו.

- לקוד סופקה פונקציית $main$ המכילה מספר בדיקות בסיסיות לקוד, הכוללות מספר בדיקות בין היתר עבור הדרישות השונות שצוינו מטה.

## איך זה מספק את הדרישות

1. $CA$ יכול לספק תעודות ל־$CA$ אחרים - בעת מתן תעודה חדשה עבור $CA$ יודלק הערך הבוליאני של $Is\ CA$ בתעודה, ובכך יסומן כי זו תעודה המשוייכת ל־$CA$ עם כל המשמעויות הנלוות לכך. חשוב לשים לב שתעודות לא ניתנות לשינוי (יוסבר בסעיף 5), ולכן אין לשום גוף את האפשרות לשנות לעצמו את הערך הבוליאני הנ"ל ובכך לחלק תעודות ללא סמכות.
2. מכניזם של ביטול תעודות - לכל אובייקט $CA$ קיימת רשימה של כל התעודות להן ביצע $Revoke$, אותה ה־$Relying\ Party$ בודק בהתאם בעת תהליך ה־$Verify$ עבור כל שלב בשרשרת. בנוסף, בעת ביטול תעודה על ידי $CA$ כלשהו, תופעל גם מתודה שתעבור ותמחק מהרשימה את כל התעודות המבוטלות תחתיו שכבר פגות תוקף.
3. $CA$ לא יוכל לראות לעולם מפתח סודי של $Entity$ ולהיפך - בשום שלב $Entity$ לא יקבל אובייקט $CA$ ולהיפך, ולכן תהיה לאף אובייקט גישה למפתח הסודי של אובייקט אחר. בפרט, כל ההתנהלות בין האובייקטים השונים תתבצע איך ורק באמצעות התעודות שלהם, שמובן שלא כוללות את המפתח הסודי.
4. ביצוע ולידציה לתעודות - ראשית, בנוגע לעניין תאריכי התעודות, נבדקת עבור כל תעודה שהיא אכן בתוקף שהוגדר עבורה בעת יצירתה. בנוסף, נבדוק באופן רקורסיבי את שרשרת התעודות על ידי כך שעבור כל $Entity$ או $CA$ ואובייקט או גוף שעליו חתם, ונוודא שאכן זהו האובייקט שחתם עליו באמצעות החתימה של האובייקט והמפתח הפומבי של החותם. בנוסף לכל זה, אנו מוודאים שאכן ראש השרשרת הוא $Root\ CA$ מורשה, על מנת לזהות תוקף שהמציא כי הוא גוף המורשה לחתום, וגם שהתעודה הנתונה לא בוטלה על ידי מייצר התעודה (הבא בשרשרת) או על ידי אותו יישות בעצמה.
5. תעודות לא ניתנות לשינוי - אם בעל תעודה יבצע לה שינוי, כמו לדוגמה גוף ישנה בעצמו כי הוא מורשה לחלק תעודות (הוא $CA$ בעצמו), בתהליך הולידציה של ה־$Relying\ Party$ השינוי יתפס ויוחזר כי השרשרת איננה תקינה,

שכן השינוי יגרום לשינוי האובייקט ה־$Serializable$ שבמקרה שלנו הוא התעודה עצמה, ומכך $rsa.validate$ יכשל.

## הקוד

```
import json
   import rsa
   from datetime import datetime, date
   YEARS_OF_VALIDITY = 1
   ROOT_VALIDITY = 10
   KEY_SIZE = 512
   class Entity:
   """
   Sign objects using their secret keys.
   """
   def __init__(self, name):
   """
   Initialize an Entity object.
   :param name:  String name of the entity.
   """
   self.name = name
   self.public_key, self.private_key = rsa.newkeys(KEY_SIZE)
   self.cert = None
   def sign(self, obj):
   """
   Method responsible for creating a signature.
   :param obj:  Object to be signed.
   :return:  Signature for the given object.
   """
   serialized_obj = json.dumps(obj, indent=4, sort_keys=True,
   default=str)
   signature = rsa.sign(str(serialized_obj).encode(),
   self.private_key,
   'SHA-256')
   return signature
   def set_cert(self, cert):
   """
   Sets new certificate for the Entity.
   :param cert:  New certificate to be set.
   """
   self.cert = cert
   def get_cert(self):
   """
   Returns the current certificate of the CA / Entity.
   """
   return self.cert
   class CA(Entity):
   """
   Issue certificates to other entities.
   """
   def __init__(self, name):
   """
   Each CA has its own public key, private key and list of all
   certificates that he has revoked.
```

```
:param name:  String name of the entity.
"""
super().__init__(name)
self.revoked = list()
self.cert = {'Name':  self.name,
'Issuer name':  None,
'Issuer cert':  None,
'Public key':  (self.public_key.n, self.public_key.e),
'Valid from':  datetime.now().replace(microsecond=0),
'Valid to':  datetime.now().replace(
year=date.today().year+ROOT_VALIDITY,
microsecond=0),
'Is CA': True} # Default cert for root CAs
def issue_cert(self, ent_name, ent_pkey, valid_from, valid_to,
is_ca=False):
"""
Issues a certificate to the given entity and signs it with the CA's
private key.
:param ent_name:  String name of the entity to certificate.
:param ent_pkey:  Public key of the entity to certificate.
:param valid_from:  Start time for the certificate.
:param valid_to:  End time of the certificate.
:param is_ca:  Boolean value whether the entity can be a CA itself.
:return:  New certificate issued for the given entity.
"""
cert = {'Name':  ent_name,
'Issuer name':  self.name,
'Issuer cert':  self.cert,
'Public key':  (ent_pkey.n, ent_pkey.e),
'Valid from':  valid_from,
'Valid to':  valid_to,
'Is CA': is_ca}
signature = self.sign(cert)
cert['Signature'] = signature
return cert
def revoke_cert(self, cert):
"""
Revoke the given certificate and update the revoked list according
to validity dates.
:param cert:  Certificate to revoke.
"""
self.revoked.append(cert)
self.update_revoked()
def update_revoked(self):
"""
Remove outdated certificates the from revocations list (in-place).
"""
to_remove = list()
cur_time = datetime.now().replace(microsecond=0)
for cert in self.revoked:
if cert['Valid to'] < cur_time:
to_remove.append(cert)
for cert in to_remove:
self.revoked.remove(cert)
```

```
def get_revoked(self):
"""
Returns the revoked certificates list of the CA.
"""
return self.revoked
class RelayingParty:
"""
Knows to verify objects signed by entities.
"""
def __init__(self, root_ca_lst, name_to_revoked):
"""
:param root_ca_lst:  List contains all root CAs certificates.
:param name_to_revoked:  Dictionary which maps a name of Entity to its
revoked list.
"""
self.root_ca_lst = root_ca_lst
self.name_to_revoked = name_to_revoked
def verify(self, obj, cert_chain):
"""
Given an object and cert chain, knows to check object is valid.
:param obj:  Object to be checked its validity.
:param cert_chain:  The end of certificate chain of the given object.
:return:  Boolean value whether the certificate chain given is valid,
with an explanation being printed.
"""
last_cert = obj
cert = cert_chain
while cert is not None:
# Validate Signature
if last_cert == obj:
signature = obj.signature
serialized_obj = json.dumps(obj.obj, indent=4, sort_keys=True,
default=str)
else:
signature = last_cert['Signature']
RelayingParty.update_cert(last_cert)
serialized_obj = json.dumps(last_cert, indent=4,
sort_keys=True, default=str)
RelayingParty.update_cert(last_cert, signature)
pub_key = rsa.PublicKey(cert['Public key'][0],
cert['Public key'][1])
try:
rsa.verify(str(serialized_obj).encode(), signature, pub_key)
except rsa.VerificationError:
print('Certificate chain is not valid.')
return False
# Validate certificate hasn't been revoked
if cert['Issuer name'] is not None:  # Not root
if cert in self.name_to_revoked[cert['Issuer name']]:
print('Certificate has been revoked.')
return False
if cert['Is CA'] and \
cert in self.name_to_revoked[cert['Name']]:
print('Certificate has been revoked.')  # Self revoked
```

```
return False
# Validate time hasn't passed or it is too early
cur_time = datetime.now().replace(microsecond=0)
if cert['Valid from'] > cur_time:
print('Certificate is not valid yet.')
return False
elif cert['Valid to'] < cur_time:
print('Certificate has expired.')
return False
last_cert = cert
cert = cert['Issuer cert']
# Verify that the certificates chain ends with a root CA
if last_cert not in self.root_ca_lst:
print('Certificate chain is invalid.')
return False
print('Given object is valid.')
return True
@staticmethod
def update_cert(cert, signature=None):
"""
Updates the certificate so that it fit for verifying, or turn it back
after the verify process.
:param cert:  Certificate to be updated.
:param signature:  Signature to be added back to cert.
:return:  The updated certificate.
"""
if not signature:
del cert['Signature']
else:
cert['Signature'] = signature
return cert
class Obj:
"""
Class represents an object to be signed.
"""
def __init__(self, obj, signature=None):
self.obj = obj
self.signature = signature
def set_signature(self, signature):
"""
Adds signature to the object.
:param signature:  Signature to be added.
"""
self.signature = signature
def main():
# PKI init
uni_ca = CA('UniverseCA')
root_ca_lst = list()
name_to_revoked = dict()
root_ca_lst.append(uni_ca.cert)
name_to_revoked[uni_ca.name] = uni_ca.get_revoked()
rp = RelayingParty(root_ca_lst, name_to_revoked)
# = Testing Variables =
start = datetime.now().replace(microsecond=0)
```

```
end = datetime.now().replace(
year=date.today().year+ROOT_VALIDITY,
microsecond=0)
# Basic Chain
print("=== Testing Basic Chain ===")
huji = Entity('HUJI')
huji.set_cert(uni_ca.issue_cert(huji.name, huji.public_key, start, end))
obj1 = Obj({'Name': 'Dor', 'Rule': 'CEO'})
obj1.set_signature(huji.sign(obj1.obj))
assert (rp.verify(obj1, huji.cert))
print("=== Finished Basic Chain ===\n")
# Check Validity date
print("=== Testing Expired Certificate ===")
expired = Entity('Expired')
expired.set_cert(uni_ca.issue_cert(expired.name, expired.public_key, start,
start.replace(second=start.second-1)))
obj5 = Obj({'Name': 'Miki', 'Rule': 'Former CEO'})
obj5.set_signature(expired.sign(obj5.obj))
assert (not rp.verify(obj5, expired.cert))
print("=== Finished Expired Certificate Test ===\n")
# Check Revocation
print("=== Testing Revocation ===")
uni_ca.revoke_cert(huji.cert)
assert (not rp.verify(obj1, huji.cert))
print("=== Finished Revocation ===\n")
# Complex Chain
print("=== Testing Complex Chain ===")
intel = CA('Intel')
intel.set_cert(uni_ca.issue_cert(intel.name, intel.public_key, start, end,
True))
mobileye = Entity('Mobileye')
mobileye.set_cert(intel.issue_cert(mobileye.name, mobileye.public_key,
start, end))
obj2 = Obj({'Name': 'Barak', 'Rule': 'CTO'})
obj2.set_signature(mobileye.sign(obj2.obj))
name_to_revoked[intel.name] = intel.get_revoked()
assert (rp.verify(obj2, mobileye.cert))
print("=== Finished Complex Chain ===\n")
# Invalid Signature
print("=== Testing Invalid Signature ===")
attacker = Entity('Attacker')
attacker.set_cert({'Entity name': attacker.name,
'Issuer name': uni_ca.name,
'Issuer cert': uni_ca.cert,
'Public key': (
attacker.public_key.n, attacker.public_key.e),
'Valid from': datetime.now().replace(microsecond=0),
'Valid to': datetime.now().replace(
year=date.today().year+10),
'Signature': 'Some fake signature'.encode(),
'Is CA': False}) # Catch if True (not in root_ca_lst)
obj3 = Obj({'Name': 'Eve', 'Rule': 'Owner'})
obj3.set_signature(attacker.sign(obj3.obj))
assert (not rp.verify(obj3, attacker.cert))
```

```
print("=== Finished Invalid Signature ===\n")
# Invalid Chain
print("=== Testing Invalid Chain ===")
assert (not rp.verify(obj2, intel.cert))
print("=== Finished Invalid Chain ===\n")
# Invalid Root CA
print("=== Testing Invalid Root ===")
invalid_root = CA('Invalid')
obj6 = Obj({'Name':  'Just', 'Rule':  'Check'})
obj6.set_signature(invalid_root.sign(obj6.obj))
name_to_revoked[invalid_root.name] = invalid_root.get_revoked()
assert (not rp.verify(obj6, invalid_root.cert))
print("=== Finished Invalid Root ===\n")
if __name__ == '__main__':
main()
```