



Sigurnost računala i podataka

Laboratorijske vježbe

Laboratorijska vježba 1

U prvoj laboratorijskoj vježbi realizirani su **man-in-the-middle** i **denial of service** napadi iskorištavanjem ranjivosti Address Resolution Protocol-a (ARP). Napadi su testirani u virtualiziranoj Docker mreži koju čine 3 virtualizirana Docker računala (eng. container): dvije žrtve imenovane kao station-1 i station-2 te napadač evil-station.

Podizanje Docker kontejnera i pokretanje interaktivnog shell-a

1. U Terminalu kreiramo osobni direktorij i pozicioniramo se u njega. U taj direktorij kloniramo sljedeći GitHub repozitorij:

```
git clone https://github.com/mcagalj/SRP-2021-22
```

2. Pozicioniramo se u odgovarajući direktorij za vježbu.

```
cd SRP-2021-22/arp-spoofing/
```

3. Pozivanjem skripte pokrećemo virtualizirano mrežno okruženje.

```
./start.sh
```

4. Provjera svih aktivnih kontejnera.

```
docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | NAMES |
|--------------|---------|---------|----------------|---------------|--------------|
| c4660c699839 | srp/arp | "bash" | 50 seconds ago | Up 48 seconds | station-2 |
| 6f11986b1c1a | srp/arp | "bash" | 50 seconds ago | Up 48 seconds | evil-station |
| 6f4a4d63bea2 | srp/arp | "bash" | 50 seconds ago | Up 49 seconds | station-1 |

5. Pokrećemo interaktivni shell na svakom od kontejnera u zasebnim prozorima Terminala.

```
docker exec -it station-naziv bash
```

6. Iz interaktivnog shell-a od station-1 provjerimo je li dohvatljiv station-2, odnosno jesu li oba na istoj mreži.

```
ping station-2
```

```
64 bytes from station-2.srp-lab (172.21.0.3): icmp_seq=1 ttl=64 time=7.01 ms
64 bytes from station-2.srp-lab (172.21.0.3): icmp_seq=2 ttl=64 time=0.229 ms
64 bytes from station-2.srp-lab (172.21.0.3): icmp_seq=3 ttl=64 time=0.140 ms
```

Provođenje napada



Man in the middle (MITM) napad opisuje situacije u kojima napadač presreće komunikaciju između računala uvjeravajući ih da ona komuniciraju direktno. Napadač je tako u mogućnosti preuzeti cijelu komunikaciju bez znanja njezinih sudionika.

Koristeći netcat uslužni program možemo razmjenjivati tekstualne poruke između virtualiziranih računala žrtvi: station-1 i station-2.

- station-1:

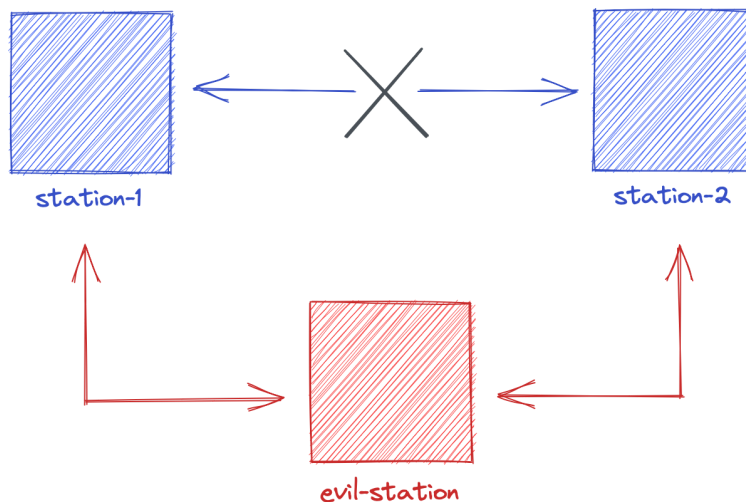
```
netcat -l -p 2000
```

- station-2:

```
netcat station-1 2000
```



ARP Spoofing je man in the middle napad u kojem napadač vezuje svoju MAC adresu s IP adresom legitimnog računala na mreži kojeg želi oponašati. Tako napadač prekida direktnu komunikaciju među računalima i presreće promet između njih.



U interaktivnom shell-u od evil-station aktiviramo arpspoof. Potrebno je definirati koja žrtva je target, a koja host. Target je onaj koji se vara: station-1, a host je onaj kojeg napadač oponaša (lažno se predstavlja kao on): station-2.

```
arpspoof -t station-1 station-2
```

U novom prozoru Terminala za evil-station pokrećemo tcpdump koji nam omogućuje praćenje paketa između računala žrtvi. Radi preglednosti filtrirali smo promet tako da se prikazuju samo tekstualne poruke razmijenjene među računalima.

```
tcpdump -X host station-1 and not arp
```

Osim praćenja prometa možemo ga u potpunosti prekinuti u oba smjera izvodeći tako denial of service napad.

```
echo 0 > /proc/sys/net/ipv4/ip_forward
```

Zaključak

U demonstriranom man in the middle napadu dolazi do narušavanja **integriteta** IP/MAC adrese. Kako smo u prvom dijelu samo analizirali promet između računala, odnosno "prisluškivali" razmijenjene poruke i nismo poduzeli nikakve mjere da manipuliramo komunikacijom, ovakav napad možemo svrstati u **pasivne napade**. U drugom dijelu, prekinuvši komunikaciju, imamo denial of service napad kod kojeg dolazi do ograničenja **dostupnosti**.

Laboratorijska vježba 2

U drugoj laboratorijskoj vježbi zadatak je riješiti presonalizirani crypto izazov, odnosno dešifrirati odgovarajući ciphertext u kontekstu simetrične kriptografije. Za svakog studenta kreirana je personalizirana, šifrirana datoteka čiji sadržaj je potrebno dešifrirati. Izazov počiva na činjenici da student nema pristup enkripcijskom ključu.

Radno okruženje

U Terminalu se pozicioniramo u osobni direktorij. Kreiramo virtualno okruženje u pythonu da ograničimo utjecaj onoga što radimo.

```
>>>python -m venv dceko
>>>cd dceko
>>>cd Scripts
>>>activate
```

Za provođenje izazova korištena je Python biblioteka **cryptography** koju je potrebno instalirati.

```
pip install cryptography
```

Testiranje sustava Fernet

Plaintext koji student treba otkriti enkriptiran je korištenjem high-level sustava za simetričnu enkripciju iz navedene biblioteke - Fernet.

```
>>>from cryptography.fernet import Fernet
>>>key = Fernet.generate_key() #generamo ključ i pohranimo ga u varijablu key
>>>f = Fernet(key)
>>>plaintext = b"Hello world" #plaintext je ono što enkriptiramo
>>>ciphertext = f.encrypt(plaintext)
>>>f.decrypt(ciphertext) #dekriptirani ciphertext daje početni plaintext
b'Hello world'
```

Preuzimanje odgovarajuće datoteke

Nazivi personaliziranih datoteka za izazov koje je potrebno preuzeti sa servera generirani su hash funkcijom po formatu **prezime_ime**.

```

from cryptography.hazmat.primitives import hashes

def hash(input):
    if not isinstance(input, bytes):
        input = input.encode()

    digest = hashes.Hash(hashes.SHA256())
    digest.update(input)
    hash = digest.finalize()

    return hash.hex()

if __name__ == "__main__":
    h = hash('ceko_david')
    print(h)

```

Izvođenjem gornjeg isječka koda dobijemo naziv svoje datoteke, u mom slučaju:

5ec9778f4669d4d6a2d91dcef88b03da4ed597fa764952445610f17628ebd3e7.encrypted

Dešifriranje crypto izazova

Preuzete datoteke enkriptirane su ključevima entropije od 22 bita, odnosno obuhvaćeni *keyspace* iznosi 2^{22} mogućih ključeva. Do ključa za dekriptiranje naše datoteke dolazimo *brute force* napadom, odnosno provjeravamo sve moguće ključeve. Dekriptirani *crypto* izazov potrebno je pohraniti u odgovarajuću datoteku.

```

import base64
from cryptography.hazmat.primitives import hashes
from cryptography.fernet import Fernet

def test_png(header):
    if header.startswith(b'\211PNG\r\n\032\n'):
        return True

def hash(input):
    if not isinstance(input, bytes):
        input = input.encode()

    digest = hashes.Hash(hashes.SHA256())
    digest.update(input)
    hash = digest.finalize()
    return hash.hex()

def brute_force():
    filename = "5ec9778f4669d4d6a2d91dcef88b03da4ed597fa764952445610f17628ebd3e7.encrypted"
    with open(filename, "rb") as file:

```

```

    ciphertext = file.read()
    ctr = 0
    while True:
        key_bytes = ctr.to_bytes(32, "big")
        key = base64.urlsafe_b64encode(key_bytes)
        if not (ctr+1) % 1000:
            print(f"[*] Keys tested: {ctr + 1}", end="\r")
        try:
            plaintext = Fernet(key).decrypt(ciphertext)
            header = plaintext[:32]
            if test_png(header):
                print(f"[+] KEY FOUND: {key}")
                with open(BINGO.png, "wb") as file:
                    file.write("plaintext")
                break
        except Exception:
            pass
        ctr += 1

if __name__ == "__main__":
    brute_force()

```

Datoteka koja se dobije nakon što smo došli do odgovarajućeg enkripcijskog ključa i dešifrirali personalizirani izazov je priložena .png slika:

Congratulations Ceko David!
You made it!

Laboratorijska vježba 3

Treća laboratorijska vježba bavi se osnovnim kriptografskim mehanizmima za autentikaciju i zaštitu integriteta poruka u praktičnom primjerima. U primjerima na vježbi korišteni su simetrični i asimetrične kripto mehanizmi: **message authentication code** (MAC) i **digitalni potpisi** zasnovani na javnim ključevima. Vježba sadrži dva izazova koja je potrebno riješiti.

Radno okruženje

U Terminalu se pozicioniramo u osobni direktorij. Kreiramo virtualno okruženje u pythonu da ograničimo utjecaj onoga što radimo - postupak je isti kao u prijašnjoj vježbi 2. Za provođenje izazova korištena je Python biblioteka **cryptography** koju je potrebno imati instaliranu.

Izazov 1

Potrebno je kreirati tekstualnu datoteku čiji integritet želimo zaštititi. Koristeći HMAC mehanizam iz python biblioteke cryptography implementiramo zaštitu integriteta pohranjene poruke. Rezultat izvršavanja donjeg koda je ispis vrijednosti MAC-a. Dobivenu vrijednost na kraju pohranimo u signature.sig datoteku.

```
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.exceptions import InvalidSignature

def generate_MAC(key, message):
    if not isinstance(message, bytes):
        message = message.encode()

    h = hmac.HMAC(key, hashes.SHA256())
    h.update(message)
    signature = h.finalize()
    return signature

if __name__ == "__main__":
    key=b"my secret password"
    with open("message.txt", "rb") as file:
        content = file.read()

    mac = generate_MAC(key, content)
    with open("message.sig", "wb") as file:
        file.write(mac)
    print(mac.hex())
```


Svakim pokretanjem rezultat je isti - ova hash funkcija je deterministička, odnosno za istu poruku uvijek generira isti MAC. Provjeru validnosti MAC-a možemo izvršiti uz pomoć sljedeće funkcije:

```
def verify_MAC(key, signature, message):
    if not isinstance(message, bytes):
        message = message.encode()

    h = hmac.HMAC(key, hashes.SHA256())
    h.update(message)
    try:
        h.verify(signature)
    except InvalidSignature:
        return False
    else:
        return True
```

Minimalnom modifikacijom sadržaja poruke ili uređivanja signature.sig datoteke u hex editoru dobivamo rezultat False - tada je došlo do narušavanja integriteta. Ovom provjerom ne možemo ustanoviti gdje je došlo do modifikacija, u samoj poruci ili vrijednosti MAC-a.

Izazov 2

Cilj ovog izazova je utvrditi vremenski ispravnu skevencu transakcija (ispravan redosljed transakcija) sa odgovarajućim dionicama. Digitalno potpisani (primjenom MAC-a) nalozi za pojedine transakcije nalaze se na lokalnom poslužitelju. Potrebno je preuzeti sve datoteke koristeći alat wget u naš aktivni direktorij. Funkcije iz prvog izazova koriste se za provjeru validnosti svake od .txt datoteka izazova.

```
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.exceptions import InvalidSignature

def verify_MAC(key, signature, message):
    if not isinstance(message, bytes):
        message = message.encode()
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(message)
    try:
        h.verify(signature)
    except InvalidSignature:
        return False
    else:
        return True
```

```

def generate_MAC(key, message):
    if not isinstance(message, bytes):
        message = message.encode()

    h = hmac.HMAC(key, hashes.SHA256())
    h.update(message)
    signature = h.finalize()
    return signature

if __name__ == "__main__":

    key = "ceko_david".encode()

    for ctr in range(1, 11):
        msg_filename = f"order_{ctr}.txt"
        sig_filename = f"order_{ctr}.sig"
        with open(msg_filename, "rb") as file:
            message = file.read()
        with open(sig_filename, "rb") as file:
            signature = file.read()

        is_authentic = verify_MAC(key, signature, message)
        print(f'Message {message.decode():>45} {"OK" if is_authentic else "NOK":<6}')
```

Rezultat izvršavanja je:

```

Message    Buy 56 shares of Tesla (2021-11-15T08:20) NOK
Message    Buy 48 shares of Tesla (2021-11-12T05:21) NOK
Message    Sell 65 shares of Tesla (2021-11-09T03:05) NOK
Message    Sell 44 shares of Tesla (2021-11-13T19:16) OK
Message    Sell 72 shares of Tesla (2021-11-11T15:22) NOK
Message    Sell 92 shares of Tesla (2021-11-12T16:31) OK
Message    Buy 21 shares of Tesla (2021-11-15T03:20) OK
Message    Sell 58 shares of Tesla (2021-11-09T10:35) OK
Message    Buy 77 shares of Tesla (2021-11-12T13:55) OK
Message    Sell 31 shares of Tesla (2021-11-12T20:44) OK
```

Provjerena je vjerodostojnost svake od transakcija, a ispis "OK" indicira vjerodostojnu transakciju. Ispravne transakcije sada se mogu organizirati u vremenski ispravnu skevencu.

Izazov 3

Ovaj izazov bavi se digitalnim potpisima u kontekstu public-key kriptografije. Cilj je odrediti autentičnu sliku koju je profesor potpisao svojim privatnim ključem. Javni ključ i

fotografije potrebne za provođenje izazova dostupne su na lokalnom poslužitelju. Korištena je Python biblioteka **cryptography**, konkretnije **RSA kriptosustav**.

Prvo učitamo ključ iz datoteke:

```
def load_public_key():
    with open("public.pem", "rb") as f:
        PUBLIC_KEY = serialization.load_pem_public_key(
            f.read(),
            backend=default_backend()
        )
    return PUBLIC_KEY
```

Učitani ključ može se ispisati pozivom gore definirane funkcije:

```
public_key = load_public_key()
print(public_key)
```

```
<cryptography.hazmat.backends.openssl.rsa._RSAPublicKey object at 0x000001F7699601C0>
```

Funkcija koju koristimo za provjeru autentičnosti kao argumente prima digitalni potpis i “poruku”, odnosno naše slike. Ispis "True" označava da je slika autentična, a ispis "False" da slika nije autentična. Barem jedna slika iz izazova mora dati pozitivan rezultat.

```
def verify_signature_rsa(signature, message):
    PUBLIC_KEY = load_public_key()
    try:
        PUBLIC_KEY.verify(
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
    except InvalidSignature:
        return False
    else:
        return True
```

```
with open("image_1.sig", "rb") as file:
    signature = file.read()
with open("image_1.png", "rb") as file:
    image = file.read()

is_authentic = verify_signature_rsa(signature, image)
print(is_authentic)
```



Da bismo izvršili provjeru autentičnosti potrebni su sam digitalni potpis i “poruka”, odnosno naše slike. Razlog tomu je način funkcioniranja provjere autentičnosti putem digitalnih potpisa - hash vrijednost slike uspoređuje se s dekriptiranom hash vrijednošću iz digitalnog potpisa.

Laboratorijska vježba 4

Laboratorijska vježba 4 pobliže nas upoznaje sa osnovnim konceptima relevantnim za sigurnu pohranu lozinki. Usporediti ćemo klasične (*brze*) kriptografske *hash* funkcije sa specijaliziranim (*sporim* i *memorijski zahtjevnim*) kriptografskim funkcijama za sigurnu pohranu zaporki i izvođenje enkripcijskih ključeva (*key derivation function (KDF)*).

Linux Hash funkcija

Prvi dio vježbe analizirali smo Linux Hash funkciju. Početna vrijednost broja iteracija za ovu funkciju postavljena je na 5000. Tesitramo vrijeme izvršavanja za tu funkciju za 5000 iteracija, ali i za 1 milijun iteracija. Parametrom **rounds** postavljamo željeni broj iteracija.

```
{
  "name": "Linux CRYPTO 5K",
  "service": lambda: linux_hash(password, measure=True)
},
{
  "name": "Linux CRYPTO 1M",
  "service": lambda: linux_hash(password, rounds=10**6, measure=True)
},
```

| Function | Avg. Time (100 runs) |
|-----------------|----------------------|
| HASH_SHA256 | 3.1e-05 |
| HASH_MD5 | 3.3e-05 |
| AES | 0.000655 |
| Linux CRYPTO 5K | 0.006402 |
| Linux CRYPTO 1M | 1.222097 |

Cilj je kriptografske hash funkcija koje se koriste pri pohrani lozinki učiniti sporijima. Parametar **rounds** iznosa 100 znači 100 iterativnih prolaza kroz hash funkciju prilikom pohrane lozinke. Pri pokušaju izrade rječnika, kojim bi se mogao ostvariti offline password napad, napadaču je potrebno puno više vremena za generiranje svih parova hash vrijednosti i lozinki koje bi sačinjavale taj rječnik.