Pwntools

1. process & remote

- process() 함수는 익스플로잇을 로컬 바이너리 대상으로 할 때 사용하는 함수이고, remote( ) 함수는 원격 서버를 대상으로 할 떄 사용하는 함수이다.
    - 전자는 보통 익스플로잇을 테스트하고 디버깅하기 위해, 그리고 후자는 대상 서버를 실제로 공격하기 위해 사용한다.

```
from pwn import *
p = process('./test')  # 로컬 바이너리 'test'를 대상으로 익스플로잇 수행
p = remote('example.com', 31337)  # 'example.com'의 31337 포트에서 실행 중인
```

2. send

- send는 데이터를 프로세스에 전송하기 위해 사용한다.
    - Pwmtools에는 관련된 다양한 함수가 정의되어 있다.

```
from pwn import *
p = process('./test')

p.send(b'A')  # ./test에 b'A'를 입력
p.sendline(b'A') # ./test에 b'A' + b'\n'을 입력
p.sendafter(b'hello', b'A')  # ./test가 b'hello'를 출력하면, b'A'를 입력
p.sendlineafter(b'hello', b'A')  # ./test가 b'hello'를 출력하면, b'A' + b'\n'을 입력
```

3. recv

- recv는 프로세스에서 데이터를 받기 위해 사용한다.
    - 마찬가지로 pwntools에는 관련된 다양한 함수가 정의되어 있다.
        - 여기서 주의해서 봐야할 것은 recv()와 recvn()의 차이점이다.
            - recv(n)은 최대 n 바이트를 받는 것이므로, 그만큼을 받지 못해도 에러를 발생시키지 않지만, recvn(n)의 경우 정확히 n바이트의 데이터를 받지 못하면 계속 기다린다.

```
from pwn import *
p = process('./test')

data = p.recv(1024)  # p가 출력하는 데이터를 최대 1024바이트까지 받아서 data에 저
data = p.recvline()  # p가 출력하는 데이터를 개행문자를 만날 때까지 받아서 data에 저
data = p.recvn(5)  # p가 출력하는 데이터를 5바이트만 받아서 data에 저장
data = p.recvuntil(b'hello')  # p가 b'hello'를 출력할 때까지 데이터를 수신하여 data에
data = p.recvall()  # p가 출력하는 데이터를 프로세스가 종료될 때까지 받아서 data에 저
```

4. packing & unpacking

- 익스플로잇을 작성하다보면 어떤 값을 리틀 엔디언의 바이트 배열로 변경하거나, 또는 역의 과정을 거쳐야 하는 경우가 자주 있다.
    - pwntools에는 관련된 함수들이 정의되어 있다.

```
#!/usr/bin/env python3
# Name: pup.py

from pwn import *

s32 = 0x41424344
s64 = 0x4142434445464748

print(p32(s32))
print(p64(s64))

s32 = b"ABCD"
s64 = b"ABCDEFGH"

print(hex(u32(s32)))
print(hex(u64(s64)))
```

- 실행 결과

```
$ python3 pup.py
b'DCBA'
```

```
b'HGFEDCBA'
0x44434241
0x4847464544434241
```

5. interactive

- 셸을 획득했거나, 익스플로잇의 특정 상황에 직접 익력을 주면서 출력을 확인하고 싶을 때 사용하는 함수이다.
    - 호출하고 나면 터미널로 프로세스에 데이터를 입력하고, 프로세스의 출력을 확인할 수 있다.

```
from pwn import *
p = process('./test')
p.interactive()
```

6. ELF

- ELF헤더에는 익스플로잇에 사용될 수 있는 각종 정보가 기록되어 있다.
    - pwntools를 사용하면 해당 정보들을 쉽게 참조할 수 있다.

```
from pwn import *
e = ELF('./test')
puts_plt = e.plt['puts'] # ./test에서 puts()의 PLT주소를 찾아서 puts_plt에 
read_got = e.got['read'] # ./test에서 read()의 GOT주소를 찾아서 read_got
```

7. context.log

- 익스플로잇에 버그가 발생하면 익스플로잇도 디버깅해야 한다.
    - pwntools에는 디버그의 편의를 돕는 로깅 기능이 이씅며, 로그 레벨은 context.log_level 변수로 조절할 수 있다.

```
from pwn import *
context.log_level = 'error' # 에러만 출력
```

```
context.log_level = 'debug' # 대상 프로세스와 익스플로잇간에 오가는 모든 데이터를
context.log_level = 'info'  # 비교적 중요한 정보들만 출력
```

8. context.arch
    - pwntools는 셸코드를 생성하거나, 코드를 어셈블, 디스어셈블하는 기능 등을 가지고 있는데, 이들은 공격 대상의 아키텍처에 영향을 받는다.
        - 그래서 pwntools는 아키텍처 정보를 프로그래머가 지정할 수 있게 하며, 해당 값에 따라 몇몇 함수들의 동작이 달라진다.

```
from pwn import *
context.arch = "amd64" # x86-64 아키텍처
context.arch = "i386"  # x86 아키텍처
context.arch = "arm"   # arm 아키텍처
```

9. shellcraft
- pwntools에는 자주 사용되는 셸 코드들이 저장되어 있어서, 공격에 필요한 셸 코드를 쉽게 꺼내쓸 수 있게 해준다.
    - 매우 편리한 기능이지만, 정적으로 생성된 코드는 셸 코드가 실행될 때의 메모리 상태를 반영하지 못한다.
        - 또한 프로그램에 따라 입력할 수 있는 셸 코드의 길이나, 구성 가능한 문자의 종류에 제한이 있을 수 있는데, 이런 조건들도 반영하기 어렵다.
            - 따라서 제약 조건이 존재하는 상황에서는 직접 셸 코드를 작성하는 것이 좋다.

```
#!/usr/bin/env python3
# Name: shellcraft.py

from pwn import *
context.arch = 'amd64' # 대상 아키텍처 x86-64

code = shellcraft.sh() # 셸을 실행하는 셸 코드
print(code)
```

- 실행 결과

```
$ python3 shellcraft.py
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push b'/bin///sh\x00' */
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    ...
    syscall`
```

10. asm

- pwntools는 어셈블 기능을 제공한다.
    - 해당 기능도 아키텍처가 중요하므로 아키텍처를 미리 지정해야 한다.

```
#!/usr/bin/env python3
# Name: asm.py

from pwn import *
context.arch = 'amd64' # 익스플로잇 대상 아키텍처 'x86-64'

code = shellcraft.sh() # 셸을 실행하는 셸 코드
code = asm(code)       # 셸 코드를 기계어로 어셈블
print(code)
```

- 실행 결과

```
$ python3 asm.py
b'jhH\xb8/bin///sPH\x89\xe7hri\x01\x01\x814$\x01\x01\x01\x011\xf6Vj\x08^H\
```