

수업 내용 정리 보고서

작성일: 2026년 6월 25일

챕터 구성

- 모의해킹 프로세스 및 단계별 핵심 보안 도구
- 파이썬 객체지향 프로그래밍(OOP)의 기초와 데이터 캡슐화
- 파이썬 고급 OOP 메커니즘 및 메모리 최적화
- 핵심 개념 검증 및 실무 적용 체크리스트

모의해킹 프로세스 및 단계별 핵심 보안 도구

모의해킹(Penetration Testing)은 단순히 시스템을 공격하는 행위를 넘어, 실제 공격자의 관점에서 취약점을 식별하고 이를 통해 발생할 수 있는 비즈니스 영향을 평가하여 대응 방안을 제시하는 체계적인 보안 프로세스입니다. 본 장에서는 모의해킹의 표준 단계와 각 단계에서 실무적으로 사용되는 핵심 도구들을 정리합니다.

1. 칼리 리눅스(Kali Linux)의 이해

모의해킹을 수행하기 위한 가장 대표적인 환경은 **칼리 리눅스(Kali Linux)**입니다. 데비안(Debian) 기반의 이 운영체제는 전 세계 보안 전문가들이 사용하는 수많은 해킹 도구들이 사전 설치 및 최적화되어 있습니다. 특히 커널 레벨에서 **패킷 주입(Packet Injection)** 기능을 지원하여 무선 네트워크 보안 진단에 탁월한 성능을 발휘합니다.

2. 모의해킹의 6단계 표준 프로세스

성공적인 모의해킹은 무분별한 공격이 아닌, 다음과 같은 정해진 절차에 따라 논리적으로 진행되어야 합니다.

- 대상 선정(Target Selection):** 공격할 범위(Scope)를 정의하고 허가된 자산인지 확인합니다.
- 정보 수집(Information Gathering):** 타겟의 IP, 도메인, 서비스 정보 등을 파악하여 공격 표면(Attack Surface)을 넓힙니다.
- 취약점 분석(Vulnerability Analysis):** 수집된 정보를 바탕으로 시스템의 보안 약점이나 설정 오류를 찾아냅니다.
- Exploit (공격):** 발견된 취약점을 실제로 이용하여 시스템 제어권을 획득하거나 데이터를 탈취합니다.
- 영향 평가(Impact Assessment):** 침투 성공 시 기업에 미칠 수 있는 실질적인 피해 규모를 측정합니다.
- 분석 및 문서화(Analysis & Documentation):** 공격 경로를 역추적하여 방어 전략을 수립하고, 발견된 모든 사항을 보고서로 작성합니다.

3. 단계별 핵심 모의해킹 도구 상세 분석

STAGE 1. 정보 수집 (Information Gathering)

공격의 성패는 얼마나 정교한 정보를 수집하느냐에 달려 있습니다. 이 단계에서는 네트워크 구조와 서비스 상태를 파악하는 것이 주 목적입니다.

- Nmap (Network Mapper):** 네트워크 스캐닝의 표준 도구입니다. 특정 IP의 열려 있는 포트와 구동 중인 서비스의 버전, 운영체제(OS)를 탐지합니다.
`$ nmap -sV -O [타겟IP]` (서비스 버전 및 OS 추정)
- Wireshark:** 네트워크 패킷 분석기(Packet Sniffer)입니다. 실시간으로 흐르는 데이터를 캡처하여 암호화되지 않은 프로토콜(HTTP, FTP 등) 내의 민감 정보를 확인합니다.
- Gobuster / FFUF:** 웹 서버의 숨겨진 디렉토리나 파일을 찾는 도구입니다. 사전 파일(Wordlist)을 이용해 관리자 페이지(/admin)나 백업 파일(.zip)을 찾아냅니다. FFUF는 Go 언어 기반으로 매우 빠른 속도를 자랑합니다.
- Maltego:** 오픈소스 인텔리전스(OSINT) 도구입니다. 도메인, 이메일, IP 간의 상관관계를 그래픽으로 시각화하여 타겟의 인적/물적 자원 관계를 분석합니다.

STAGE 2. 취약점 분석 (Vulnerability Analysis)

수집된 정보를 바탕으로 자동화된 스캐너나 수동 분석을 통해 보안 구멍을 찾는 단계입니다.

- Nikto:** 웹 서버의 설정 오류, 오래된 버전, 위험한 파일을 빠르게 스캔하는 웹 취약점 스캐너입니다.
- Burp Suite:** 웹 해킹의 핵심 도구로, 브라우저와 서버 사이의 통신을 가로채(Intercept) 패킷을 변조함으로써 SQL 인젝션이나 XSS 취약점을 진단합니다.
- WPScan:** 워드프레스(WordsPress) 전용 스캐너로, 취약한 플러그인이나 테마, 사용자 계정 정보를 추출합니다.
- 암호 크래킹>Password Cracking):** 탈취한 해시(Hash) 값을 원문으로 복원하는 단계입니다.
 - John the Ripper:** CPU를 사용하여 오프라인 해시를 크래킹하는 전통적인 도구입니다.
 - Hashcat:** GPU의 병렬 연산 능력을 사용하여 초고속으로 암호를 깨부수는 최신 도구입니다.

STAGE 3. 모의 침투 및 공격 (Exploit)

실제로 시스템의 방어진을 뚫고 들어가는 단계입니다.

- Metasploit Framework:** 전 세계의 취약점 공격 코드(Exploit)와 페이로드(Payload)가 집약된 거대한 플랫폼입니다. `msfconsole`을 통해 공격을 자동화합니다.
- Hydra:** 로그인 창을 대상으로 아이디와 비밀번호를 무차별 대입(Brute-Force)하는 툴입니다.
- SQLmap:** SQL 인젝션 취약점을 자동으로 탐지하고 데이터베이스 내용을 통째로 덤프(Dump)하는 특화 도구입니다.
- BeEF / SET:** 사용자 측면의 공격을 수행합니다. BeEF는 웹 브라우저를 제어하고, SET(Social Engineering Toolkit)은 피싱 사이트 복제 등을 통해 사람의 심리를 이용합니다.

STAGE 4 & 5. 권한 상승 및 사후 행위 (Post-Exploit & Pivoting)

최초 침투에 성공한 후, 관리자 권한을 얻거나 내부망의 다른 시스템으로 침투 범위를 넓히는 과정입니다.

- **LinPEAS / WinPEAS**: 시스템 내부의 설정 오류를 찾아내어 최고 관리자(Root/Admin) 권한으로 올라갈 수 있는 경로를 알려주는 스크립트입니다.
- **Mimikatz**: 윈도우 메모리(LSASS)에서 로그인된 사용자의 평문 비밀번호나 해시 값을 탈취하는 강력한 도구입니다.
- **Responder**: 내부 네트워크(LAN)에서 가짜 응답을 보내 사용자의 인증 정보(NetNTLM 해시)를 가로챍니다.
- **NetExec (구 CrackMapExec)**: 획득한 권한을 이용해 내부망의 수많은 PC에 동일 권한이 있는지 확인하고 명령을 실행하는 자동화 도구입니다.
- **Chisel / Ligolo-ng**: 네트워크 터널링(Pivoting) 도구입니다. 외부에서 접근 불가능한 내부 망(DMZ 뒤쪽 등)으로 공격 트래픽을 전달하기 위한 통로를 구축합니다.

4. 무선 네트워크 특화 (Wireless Hacking)

무선 환경에서는 패킷 주입 기능을 활용한 **Aircrack-ng Suite**가 핵심입니다. 모니터 모드로 전환하여 주변 패킷을 수집하고, 가짜 인증 해제(Deauth) 패킷을 주입해 사용자를 튜터낸 뒤, 재접속 시 발생하는 핸드셰이크(Handshake) 파일을 크래킹하여 Wi-Fi 비밀번호를 탈취합니다.

[실무 및 시험 핵심 요약]

공격 단계	핵심 목표	대표 도구	실무 판단 기준
정보 수집	공격 대상 식별	Nmap, Gobuster	열린 포트와 서비스 버전이 알려진 취약점과 매칭되는가?
취약점 분석	약점 발견	Burp Suite, Nikto	입력값 검증 미흡이나 잘못된 설정이 존재하는가?
공격 (Exploit)	권한 획득	Metasploit, SQLmap	취약점을 통해 시스템 쉘(Shell)을 획득할 수 있는가?
사후 행위	권한 상승/확산	Mimikatz, Chisel	관리자 권한을 얻었는가? 내부망의 다른 서버로 이동 가능한가?

⚠ **주의사항**: 모든 모의해킹 활동은 반드시 사전에 승인된 대상과 범위 내에서만 수행되어야 합니다. 허가받지 않은 대상에 대한 스캐닝 및 공격 시도는 법적 처벌의 대상이 됩니다.

[자가 점검 질문]

- Nmap에서 서비스 버전을 확인하기 위해 사용하는 옵션은 무엇인가?
- 웹 애플리케이션의 패킷을 중간에서 가로채고 변조하기 위해 가장 적합한 도구는?
- 시스템 침투 후, 내부망의 다른 PC로 이동하기 위해 필요한 기술(Pivoting)을 지원하는 도구는 무엇인가?
- 일반 사용자 권한을 최고 관리자(Root) 권한으로 격상시키기 위해 사용하는 자동화 스크립트는?

파이썬 객체지향 프로그래밍(OOP)의 기초와 데이터 캡슐화

현대 소프트웨어 개발에서 복잡한 시스템을 효율적으로 관리하기 위해 사용하는 핵심 패러다임이 바로 **객체지향 프로그래밍(Object-Oriented Programming, OOP)**입니다. 파이썬은 "모든 것이 객체(Everything is an object)"라는 철학을 가진 언어로, 데이터와 그 데이터를 다루는 함수를 하나의 단위로 묶어 관리하는 데 탁월한 성능을 발휘합니다.

1. 클래스와 인스턴스: 설계도와 실체

객체지향의 가장 기초는 클래스와 인스턴스의 관계를 이해하는 것에서 시작됩니다.

- **클래스(Class)**: 객체를 만들기 위한 '설계도' 또는 '틀'입니다. 어떤 데이터(변수)를 가지고 어떤 동작(메서드)을 할 것인지 정의합니다.
- **인스턴스(Instance)**: 클래스라는 설계도를 바탕으로 메모리에 실제로 구현된 '객체'입니다. 설계도는 하나지만, 이를 통해 만들어진 인스턴스는 각각 독립적인 데이터를 가질 수 있습니다.

예를 들어, '붕어빵 틀'은 클래스이고, 그 틀에서 구워져 나온 실제 '붕어빵'들은 각각 다른 내용물(팥, 슈크림 등)을 가진 인스턴스라고 비유할 수 있습니다.

2. 객체의 생명 주기와 매직 메서드(Magic Method)

파이썬에는 사용자가 직접 호출하지 않아도 인터프리터가 특정 상황에서 자동으로 호출하는 특별한 함수들이 있습니다. 이를 **매직 메서드(Magic Method)** 또는 **던더(Dunder - Double Under의 줄임말)** 메서드라고 부릅니다.

2.1 생성자(__init__)와 소멸자(__del__)

객체가 생성되고 소멸되는 시점을 제어하여 리소스를 관리합니다.

- **__init__(self, ...)**: 객체가 생성(인스턴스화)될 때 자동으로 호출되는 생성자입니다. 객체의 초기 상태(변수 값 등)를 설정하는 역할을 합니다.
- **__del__(self)**: 객체가 메모리에서 완전히 사라질 때 호출되는 소멸자입니다. 파이썬의 '참조 횟수 계산(Reference Counting)' 방식에 의해 해당 객체를 가리키는 변수가 없어지고 참조 횟수가 0이 되면 가비지 컬렉터에 의해 호출됩니다.

2.2 모듈 보호를 위한 __name__ == '__main__'

파이썬 파일이 직접 실행되는 경우와 다른 파일에서 `import`되어 부품으로 사용되는 경우를 구분하기 위한 방어벽 코드입니다. 이 구문을 사용하면 모듈을 `import`할 때 의도치 않게 테스트 코드나 메인 로직이 실행되는 것을 방지할 수 있습니다.

3. 메서드의 종류와 self의 역할

클래스 내부의 함수를 메서드라고 부르며, 파이썬에서는 인자 전달 방식에 따라 세 가지로 엄격히 구분됩니다.

구분	선언 방식	첫 번째 인자	주요 용도
인스턴스 메서드	<code>def method(self, ...)</code>	<code>self</code> (인스턴스 자신)	개별 객체의 상태(인스턴스 변수)를 읽거나 수정
클래스 메서드	<code>@classmethod</code> <code>def method(cls, ...)</code>	<code>cls</code> (클래스 자체)	클래스 변수 제어 및 팩토리 메서드 구현
정적 메서드	<code>@staticmethod</code> <code>def method(...)</code>	없음	클래스와 논리적으로 묶여 있으나 상태를 바꾸지 않는 유틸리티 기능

왜 self를 사용하는가?

인스턴스 메서드 호출 시, 파이썬은 내부적으로 객체.`메서드` (객체) 형태로 변환하여 실행합니다. 즉, 메서드가 "어떤 객체의 데이터를 건드려야 하는지"를 알기 위해 객체 자신을 첫 번째 인자로 전달받아야 하며, 이를 위해 메서드 정의 시 반드시 `self`를 명시해야 합니다.

4. 데이터 캡슐화(Encapsulation)와 속성 제어

캡슐화란 객체의 내부 데이터(변수)를 외부에서 함부로 수정하지 못하도록 보호하고, 허용된 통로를 통해서만 접근하게 만드는 보안 기법입니다.

4.1 접근 제한 규칙

- **`_variable` (Protected):** 관례적으로 "내부에서만 사용하니 외부에서 직접 건드리지 마시오"라는 의미를 전달합니다.
- **`__variable` (Private):** 파이썬의 네임 망글링(Name Mangling)을 통해 외부 접근을 더욱 까다롭게 제한합니다.

4.2 Pythonic한 Getter와 Setter: `@property`

자바(Java)와 같은 언어에서는 `getName()`, `setName()`과 같은 명시적인 함수를 사용하지만, 파이썬은 `@property` 데코레이터를 사용하여 **변수처럼 보이면서도 내부적으로는 함수가 작동하는** 깔끔한 방식을 권장합니다.

```
class Person:
    def __init__(self, name):
        self._name = name # 보호된 변수

    @property
    def name(self):
        """Getter: 변수처럼 접근할 때 호출됨"""
        return self._name

    @name.setter
    def name(self, value):
        """Setter: 값을 대입할 때 호출되며, 데이터 검증 가능"""
        if not value:
            raise ValueError("이름은 빈 문자열일 수 없습니다.")
        self._name = value

# 사용 예시
p = Person("철수")
print(p.name) # p.name()이 아니라 변수처럼 접근 (Getter 작동)
p.name = "영희" # p.set_name("영희")가 아니라 대입 연산자 사용 (Setter 작동)
```

캡슐화의 이점:

1. **데이터 무결성:** Setter 내부에서 조건문을 통해 잘못된 데이터(예: 빈 문자열, 음수 나이 등)가 저장되는 것을 차단할 수 있습니다.
2. **하위 호환성 유지:** 처음에는 일반 변수로 설계했다가 나중에 검증 로직이 필요해져 `@property`로 바뀌도, 이 클래스를 사용하는 외부 코드는 수정할 필요가 없습니다.

5. 고급 객체지향 및 메모리 최적화

5.1 상속(Inheritance)과 다형성(Polymorphism)

부모 클래스의 기능을 자식 클래스가 물려받는 것을 상속이라 하며, `super()`를 통해 부모의 생성자나 메서드를 호출할 수 있습니다. 이때 자식 클래스가 부모의 메서드를 자신의 목적에 맞게 재정의하는 것을 **메서드 오버라이딩(Method Overriding)**이라고 하며, 이를 통해 하나의 인터페이스로 다양한 동작을 수행하는 다형성을 구현합니다.

5.2 메모리 효율을 위한 __slots__

파이썬 객체는 기본적으로 새로운 속성을 자유롭게 추가할 수 있도록 내부적으로 __dict__ 라는 딕셔너리 구조를 사용합니다. 하지만 수만 개의 객체를 생성해야 하는 환경에서는 이 딕셔너리가 엄청난 메모리를 점유하게 됩니다.

이때 __slots__를 사용하여 "이 클래스는 지정된 변수만 가질 수 있다"고 선언하면, __dict__가 제거되어 메모리 사용량을 극적으로 줄일 수 있습니다. 단, 이 경우 실시간으로 새로운 속성을 추가하는 동적 속성 할당은 불가능해집니다.

실행 결과 해석: 객체의 실행 흐름

클래스 정의와 인스턴스 생성, 메서드 호출, 그리고 메모리 해제 과정에서 나타나는 출력 결과를 통해 객체의 생명 주기를 확인할 수 있습니다.

```
# 실행 결과 예시
A 생성자
<__main__.A object at 0x7f1d...>
A 소멸자
static c
끝
```

위 출력 결과는 다음과 같은 단계로 진행된 것입니다.

1. `a = A():` 인스턴스가 생성되며 `__init__`이 호출되어 "A 생성자"가 출력됩니다.
2. `a.a():` 인스턴스 메서드가 실행되며, `self`를 통해 전달된 객체의 메모리 주소가 출력됩니다.
3. `del a:` 참조가 끊겨 객체가 소멸되면서 `__del__`이 호출되어 "A 소멸자"가 출력됩니다.
4. `A.c():` 정적 메서드는 인스턴스 없이 클래스 이름으로 바로 호출되어 "static c"를 출력합니다.

주의사항: 클래스 바로 아래에 변수를 선언하면(예: `class A: b = 1`), 이는 인스턴스 변수가 아닌 모든 인스턴스가 공유하는 **클래스 변수(Static 변수)**가 됩니다. 객체마다 독립적인 값을 가져야 한다면 반드시 생성자 `__init__` 내부에서 `self`.변수명으로 선언해야 합니다.

학습 점검 리스트

- [] `self`가 인스턴스 메서드에서 왜 필수적인지 설명할 수 있는가?
- [] 클래스 변수와 인스턴스 변수의 메모리 적재 시점 및 공유 여부 차이를 아는가?
- [] `@property`를 사용하는 것이 단순 메서드 호출보다 왜 '파이썬다운(Pythonic)' 방식인가?
- [] `__slots__`를 사용할 때 얻을 수 있는 이득과 포기해야 하는 기능은 무엇인가?
- [] `@staticmethod`와 `@classmethod`의 결정적인 차이(인자로 받는 값)를 구분할 수 있는가?

파이썬 고급 OOP 메커니즘 및 메모리 최적화

모의해킹 자동화 프로그램을 개발할 때, 단순히 코드를 나열하는 방식은 기능이 확장될수록 유지보수가 불가능해지는 한계가 있습니다. 이를 극복하기 위해 파이썬의 **객체지향 프로그래밍(Object-Oriented Programming, OOP)** 개념을 이해하고, 메모리 효율성과 데이터 무결성을 고려한 고급 메커니즘을 적용하는 것이 필수적입니다. 본 장에서는 파이썬 객체의 생명 주기부터 메모리 최적화를 위한 고급 기법까지 심도 있게 다룹니다.

1. 객체의 생명 주기와 메서드의 분류

파이썬에서 모든 것은 객체(Object)입니다. 클래스(Class)가 객체를 만들기 위한 '설계도'라면, 인스턴스(Instance)는 그 설계도를 바탕으로 메모리에 실제로 구현된 '실체'를 의미합니다.

1.1 생성자와 소멸자: 객체의 탄생과 죽음

객체가 생성되고 소멸되는 과정은 특수한 **매직 메서드(Magic Method)** 또는 **던더(Dunder - Double Under score)** 메서드에 의해 관리됩니다.

- `__init__(self, ...)` (생성자): 인스턴스가 생성될 때 자동으로 호출됩니다. 객체가 가져야 할 초기 상태(인스턴스 변수)를 설정하는 데 사용됩니다.
- `__del__(self)` (소멸자): 객체의 참조 횟수(Reference Count)가 0이 되어 메모리에서 해제될 때 호출됩니다. 자원 해제 등의 정리 작업을 수행합니다.

1.2 인스턴스 메서드와 self의 역할

클래스 내부의 일반적인 함수는 **인스턴스 메서드(Instance Method)**라고 부릅니다. 이 메서드들은 반드시 첫 번째 파라미터로 `self`를 가져야 합니다. `self`는 메서드를 호출한 **객체 자신**을 가리키며, 이를 통해 객체 내부의 변수나 다른 메서드에 접근할 수 있습니다.

실행 메커니즘의 이해:

사용자가 `a.a()`라고 호출하면, 파이썬 인터프리터는 내부적으로 `A.a(a)`와 같이 동작합니다. 즉, 객체 자신을 메서드의 첫 번째 인자로 자동으로 전달하는 것입니다.

1.3 메서드의 세 가지 종류 비교

파이썬은 사용 목적에 따라 메서드를 세 가지 유형으로 구분하여 제공합니다. 이를 적절히 사용하는 것이 설계의 핵심입니다.

구분	선언 방식	첫 번째 인자	주요 용도
인스턴스 메서드	<code>def method(self)</code>	<code>self</code> (객체 자신)	개별 객체의 상태(인스턴스 변수)를 읽거나 수정
클래스 메서드	<code>@classmethod</code> <code>def method(cls)</code>	<code>cls</code> (클래스 자체)	클래스 변수를 제어하거나 객체 생성의 다른 통로(Factory) 제공
정적 메서드	<code>@staticmethod</code> <code>def method()</code>	없음	클래스와 논리적으로 연관되나 상태를 변경하지 않는 유틸리티 기능

```
# 메서드 종류 예시 코드
class Example:
    def instance_method(self):
        return "인스턴스 변수에 접근합니다."
```

```
# 실제 실행 결과 분석
A 생성자
<__main__.A object at 0x7f1d...>
A 소멸자
static c
끝
```

2. 데이터 캡슐화와 Pythonic한 접근 (Getter/Setter)

모의해킹 툴 제작 시, 공격 대상의 IP 주소나 인증 정보와 같은 중요한 변수가 외부에서 실수로 변경되는 것을 막아야 합니다. 이를 위해 **캡슐화(Encapsulation)**를 사용합니다.

2.1 정보 은닉 (Protected & Private)

- **`_variable` (Protected):** 관습적으로 "이 변수는 내부용이니 직접 수정하지 마시오"라는 신호를 보냅니다.
- **`__variable` (Private):** 파이썬 인터프리터 차원에서 이름을 변형하여 외부에서 직접 접근하는 것을 더 강력하게 제한합니다.

2.2 `@property`를 이용한 Getter와 Setter

자바(Java)와 같은 언어에서는 `get_name()`, `set_name()` 식의 메서드를 명시적으로 호출해야 하지만, 파이썬은 `@property` 데코레이터를 사용하여 변수처럼 직관적으로 접근하면서도 내부적으로는 로직을 실행하는 **Pythonic**한 방식을 권장합니다.

```
class Person:
    def __init__(self, name):
        self._name = name # 내부 변수

    @property
    def name(self):
        """Getter: 변수처럼 접근하면 실행됨"""
        return self._name

    @name.setter
    def name(self, value):
        """Setter: 대입 연산자(=) 사용 시 실행되며 데이터 검증 수행"""
        if not value:
            raise ValueError("이름은 빈 문자열일 수 없습니다.")
        self._name = value

p = Person("철수")
print(p.name) # Getter 호출 (p.name()이 아님에 주의)
p.name = "영희" # Setter 호출
```

이 방식의 장점:

1. **데이터 무결성:** Setter 내부에서 if문을 통해 잘못된 값이 들어오는 것을 차단할 수 있습니다.
2. **하위 호환성:** 처음에는 일반 변수로 설계했다가, 나중에 검증 로직이 필요해져서 Getter/Setter로 변경하더라도 외부 코드를 수정할 필요가 없습니다.

3. 상속과 다형성: 코드의 확장성 확보

다양한 공격 기법을 구현할 때, 공통적인 틀을 유지하면서 세부 동작만 바꾸는 기법이 필요합니다.

3.1 상속(Inheritance)과 super()

기존 클래스(부모)의 기능을 물려받아 새로운 클래스(자식)를 만드는 것입니다. 자식 클래스에서 `super()`를 호출하면 부모 클래스의 생성자나 메서드를 호출하여 부모가 가진 자산(변수, 기능)을 그대로 가져올 수 있습니다.

3.2 메서드 오버라이딩(Method Overriding)

부모로부터 물려받은 메서드를 자식 클래스의 목적에 맞게 재정의하는 것입니다. 이를 통해 동일한 함수 이름으로 각기 다른 동작을 수행하게 하는 **다형성(Polymorphism)**을 구현할 수 있습니다. 예를 들어, `Scanner` 클래스의 `scan()` 메서드를 `StealthScanner`가 재정의하여 우회 기술을 적용하는 식입니다.

4. 메모리 구조와 성능 최적화

대규모 네트워크 스캔과 같이 수만 개의 객체를 생성해야 하는 상황에서는 메모리 관리가 성능의 핵심입니다.

4.1 클래스 변수(Static) vs 인스턴스 변수

클래스 바로 아래에 선언된 변수는 **클래스 변수**로, 해당 클래스로 생성된 모든 인스턴스가 공유하는 단 하나의 메모리 공간을 가집니다. 반면, `self.` 변수로 선언된 것은 각 객체마다 별도로 존재하는 **인스턴스 변수**입니다.

```
class A:
    b = 1 # 클래스 변수 (모든 인스턴스가 공유)
    def __init__(self, name):
        self.name = name # 인스턴스 변수 (객체마다 개별 존재)
```

4.2 __dict__와 __slots__를 이용한 최적화

- **__dict__ (동적 속성):** 파이썬 객체는 실시간으로 새로운 변수를 추가할 수 있도록 내부적으로 딕셔너리 형태의 `__dict__`를 사용합니다. 이는 매우 편리하지만, 많은 메모리를 소비합니다.
- **__slots__ (메모리 고정):** 클래스 정의 시 `__slots__ = ['name', 'age']`와 같이 사용할 변수를 미리 지정하면, 객체 내부의 `__dict__`가 사라지고 지정된 공간만 사용하게 됩니다.

주의사항: `__slots__`를 사용하면 메모리는 획기적으로 줄어들지만, **실시간으로 새로운 속성을 추가하는 동적 할당이 불가능**해집니다. 따라서 객체의 구조가 명확할 때만 사용해야 합니다.

학습 점검 체크리스트

- [] `self`를 메서드의 첫 번째 인자로 놓지 않았을 때 발생하는 오류의 원인을 설명할 수 있는가?
- [] `@property`를 사용했을 때와 일반 메서드를 사용할 때의 호출 문법 차이를 아는가?
- [] 클래스 변수와 인스턴스 변수가 메모리에 저장되는 위치와 공유 범위의 차이를 이해했는가?
- [] 수만 개의 객체를 생성할 때 메모리 낭비를 줄이기 위해 `__slots__`를 고려해야 하는 이유를 설명할 수 있는가?
- [] `super()`를 사용하여 부모 클래스의 초기화 로직을 호출하는 것이 왜 중요한지 알고 있는가?

핵심 개념 검증 및 실무 적용 체크리스트

본 장은 학습한 모의해킹의 단계별 프로세스에 따른 툴 운용 능력과, 이를 자동화하기 위한 파이썬 객체지향 프로그래밍(OOP) 설계 역량을 통합적으로 검증하기 위해 작성되었습니다. 단순 암기를 넘어, **"특정 상황에서 어떤 도구를 선택할 것인가?"**와 **"공격 프로그램을 설계할 때 어떤 구조를 가져야 하는가?"**에 대한 실무적 판단 기준을 제공합니다.

1. 모의해킹 프로세스 및 툴 운용 검증

모의해킹은 단편적인 공격의 나열이 아닌, 정교한 단계(Stage)의 흐름을 따릅니다. 각 단계의 목적을 달성하지 못한 채 다음 단계로 넘어가는 것은 공격의 성공률을 낮추는 흔한 실수입니다.

[STAGE 1 & 2] 정보 수집 및 취약점 분석

- **대상 식별 능력:** 타겟의 IP, 열린 포트, 운영체제(OS), 서비스 버전을 정확히 파악했는가? (Nmap의 `-sV`, `-O` 옵션 활용 능력)
- **숨겨진 자원 탐색:** 웹 서비스의 메인 화면 외에 관리자 페이지(`/admin`)나 백업 파일(`.env`, `.zip`)을 찾아냈는가? (Gobuster, FFUF 활용)
- **네트워크 흐름 파악:** 암호화되지 않은 프로토콜(HTTP, FTP 등)을 통해 민감한 데이터가 평문으로 노출되는지 확인했는가? (Wireshark 활용)
- **웹 취약점 진단:** 웹 서버 자체의 설정 오류(Nikto)와 애플리케이션 레벨의 로직 취약점(Burp Suite를 통한 패킷 변조)을 구분하여 진단했는가?

[STAGE 3 & 4] 공격 수행 및 권한 상승

- **공격 경로 최적화:** 알려진 취약점(CVE)이 있다면 Metasploit을, 로그인 창이 있다면 Hydra를, DB 입력값 검증이 미흡하다면 SQLmap을 선택하는 등 상황에 맞는 툴을 선정했는가?
- **무선 환경 대응:** 주변 Wi-Fi 신호를 캡처하고 핸드셰이크(Handshake)를 탈취하여 암호화를 해제할 준비가 되었는가? (Aircrack-ng Suite 활용)
- **사후 권한 확보(Post-Exploit):** 일반 사용자 권한 획득 후, 시스템 내부의 설정 오류를 찾아 최고 관리자(Root/Admin) 권한으로 상승시킬 경로를 확보했는가? (LinPEAS/WinPEAS 활용)
- **자격 증명 탈취:** 메모리 내에 잔류하는 평문 비밀번호나 해시 값을 추출하여 내부망 확산(Lateral Movement)의 발판을 마련했는가? (Mimikatz 활용)

2. 파이썬 자동화 도구 설계를 위한 OOP 검증

모의해킹 툴을 파이썬으로 직접 개발할 때, 코드의 확장성과 유지보수성을 결정짓는 핵심은 객체지향 설계 능력입니다.

[데이터 보호 및 무결성] 캡슐화(Encapsulation)

- **접근 제어 이해:** 변수명 앞에 언더바(_, __)를 사용하여 외부로부터 보호해야 할 내부 상태를 적절히 격리했는가?
- **데이터 검증(Validation):** @property와 @setter를 사용하여, 공격 대상의 IP 주소나 포트 번호가 잘못된 형식으로 입력되는 것을 방지하는 로직을 구현했는가?
- **하위 호환성 고려:** 처음에는 일반 변수로 설계했더라도, 나중에 검증 로직이 필요할 때 외부 코드를 수정하지 않고 Getter/Setter로 전환할 수 있는 구조인가?

[코드 구조 및 확장성] 메서드와 상속

- **메서드 유형의 적절한 선택:**
 - 인스턴스의 개별 상태(예: 특정 타겟의 IP)를 다룰 때는 **인스턴스 메서드(self)**를 사용하는가?
 - 클래스 전체의 설정이나 공통 데이터를 다룰 때는 **클래스 메서드(cls)**를 사용하는가?
 - 클래스 내부에서 논리적으로 연관되어 있으나 상태 변경이 필요 없는 유틸리티 함수는 **정적 메서드(staticmethod)**로 정의했는가?
- **다형성(Polymorphism) 구현:** Scanner라는 부모 클래스를 두고, WebScanner와 NetworkScanner가 각각 scan() 메서드를 **오버라이딩(Overriding)**하여 동작하도록 설계했는가?
- **코드 재사용:** 자식 클래스에서 부모의 기능을 그대로 쓰면서 확장할 때 super()를 사용하여 부모의 생성자(__init__)를 올바르게 호출했는가?

[시스템 최적화] 메모리 및 생명 주기

- **메모리 효율성:** 수만 개의 패킷이나 타겟 데이터를 처리해야 하는 대규모 스캐너를 설계할 때, __dict__의 오버헤드를 줄이기 위해 __slots__를 사용하여 메모리를 최적화했는가?
- **변수 선언 위치 주의:** 클래스 바로 아래에 변수를 선언하여 의도치 않게 모든 인스턴스가 공유하는 **클래스 변수(Static 변수)**가 되는 실수를 범하지 않았는가? (인스턴스 고유 변수는 반드시 __init__ 내부에서 self. 변수명으로 선언해야 함)

3. 실무 통합 체크리스트 (Summary)

모의해킹 전문가이자 툴 개발자로서 갖추어야 할 최종 판단 기준입니다.

구분	검증 항목	실패 시 나타나는 현상 (Common Mistakes)
해킹 프로세스	단계별 연계성 확인	정보 수집 부족으로 공격 단계에서 타겟 서비스 탐지에 실패함.
툴 선정	상황 적합성 판단	웹 취약점을 찾기 위해 네트워크 스캐너(Nmap)만 반복 사용하는 등 비효율 발생.
Python 설계	캡슐화 및 검증	잘못된 파라미터(예: 유효하지 않은 포트)가 전달되어 프로그램이 예기치 않게 종료됨.
Python 설계	메서드 정의 (self/cls)	인스턴스 변수에 접근하려 할 때 TypeError가 발생하거나, 모든 객체가 동일한 값을 공유함.
시스템 최적화	메모리 관리 및 성능	대규모 스캔 시 메모리 사용량이 급증하여 시스템이 느려지거나 프로그램이 강제 종료됨.

실무 팁: 자동화 프로그램을 만들 때는 "내가 사용하는 툴의 명령어를 어떻게 클래스의 메서드로 추상화할 것인가?"를 먼저 고민하십시오. 예를 들어, Nmap을 실행하는 클래스를 만들 때, 실행 결과(Output)를 객체의 속성으로 저장하고, 이를 분석하는 메서드를 분리하는 것이 좋은 설계의 시작입니다.