Desine Pattern & Security (디자인 패턴 & 보안)

사전 발표 준비 및 전체적인 이론 학습

디자인 패턴

디자인 패턴

자주 발생하는 문제를 정형화시켜 다음에 이를 방지하는 "규약" 형태의 개념

개념이므로 다음 소개할 방법의 코드가 무조건적으로 동작하는 것이 아님. 방법론에 가까움

(일단 돌아가는 코드는 건들지 말자. 라는 생각을 기반으로 함)

참고 자료 : https://refactoring.guru/ko/design-patterns/python

Algorithm

(문제 Solving 관점)
어려운 문제를 해결하기 위한 일련의 과정

Code Level

→ 요리 방법을 적어둔 요리책

Desing Pattern

(Software Reuse 관점)
자주 발생하는 문제에 대한 해결책

설계 Level

→ 문제 해결을 위한 청사진

두 개념 모두 문제를 해결하기 위한 일반적인 방법.

Desing Pattern

긍정적인 관점

이미 많이 발견된 문제를 해결하기 위해 검증된 해결책 등의 방법론

의사소통에서 사용할 수 있는 도구

방법론이므로 잘만 사용한다면
기존 코드의 재사용성을 높일 수 있음

부정적인 관점

오래된 방법론이어서 오히려 최신 프로그래밍언어에서는 기본으로 제공해주는 경우도 있음

많은 개발자는 이를 남용하는데, 간단한 코드가 오히려 어려워지고, 별도의 객체를 생성하는 상황이 발생

Design Pattern을 부정적으로 보는 기업도 있으므로, 적당히 개념만 알고 있기를 추천.

Desing Pattern

Creational Pattern (생성 패턴)

: 객체 생성 방식에 대한 방법으로, 생성과 사용을 분리

Structural Pattern (구조 패턴)

: 클래스, 객체를 구조화하여 구조를 유연하고 더 큰 구조로 조립

Behavioral Pattern (행위 패턴)

: 객체 간의 상호작용, 책임 할당

Singleton

: 특정 클래스의 인스턴스가 하나만 생성되도록 제한
→ 여러 번 접근할 경우, 모두 하나의 인스턴스로 접근되도록

```
class Singleton :
    _instance = None

    def __new__ (cls) :

        print("__new__ is called ₩n")

        if not cls._instance :
            cls._instance = super().__new__(cls)
            print("Instance is created ₩n")

        else : print("Instance is already called ₩n")

        return cls._instance

    def __init__ (self) :

        print("__init__ is called ₩n")
```

```
if (__name__ == "__main__") :

    s1 = Singleton()
    s2 = Singleton()

    print(s1 is s2)
```

[Result]
__new__ is called

Instance is created

__init__ is called
__new__ is called

Instance is already called

__init__ is called
True

Factory

: 객체 생성 로직을 서브클래스에 위임하여 캡슐화함.
→ 구체적인 클래스를 몰라도 객체 생성

```python
from abc import ABC, abstractmethod

# Product Interface
class Phone (ABC) :
    @abstractmethod
    def specifications (self) -> str :
        pass

# Concrete Product
class Apple (Phone) :
    def specifications (self) -> str :
        return "Apple iPhone"

class Samsung (Phone) :
    def specifications (self) -> str :
        return "Samsung Galaxy"
```

```python
# Creator Interface
class PhoneFactory (ABC) :
    @abstractmethod
    def create_phone (self) -> Phone :
        pass

# Concreate Creator
class AppleFactory (PhoneFactory) :
    def create_phone (self) -> Phone :
        return Apple()

class SamsungFactory (PhoneFactory) :
    def create_phone (self) -> Phone :
        return Samsung()
```

```python
if (__name__ == "__main__") :

    apple = AppleFactory().create_phone()
    print(f"Created : {apple.specifications()}")

    samsung = SamsungFactory().create_phone()
    print(f"Created : {samsung.specifications()}")
```

[Result]
Created : Apple iPhone
Created : Samsung Galaxy

Adapter

: 서로 호환되지 않는 인터페이스를 연결
→ Client는 동일한 방식으로 접근

```
# client interface
class Target :
    def request (self) :
        return "Target"

# redefined interface
class Adaptee :
    def request (self) :
        return "Adaptee"

# connect interface
class Adapter (Target) :
    def __init__ (self, adaptee) :
        self.adaptee = adaptee

    def request (self) :
        return f"Adapter : {self.adaptee.request()}"
```

```
if (__name__ == "__main__") :

    adaptee = Adaptee()
    adapter = Adapter(adaptee)
    print(adapter.request())
```

[Result]
Adapter : Adaptee

Decorator

: 기존 객체에 동적으로 기능 추가
→ main 객체는 유지하고 기능을 확장

```
class Component :
    def operation (self) :
        return "Component"

class Decorator (Component) :
    def __init__ (self, component) :
        self._component = component

    def operation (self) :
        return f"Decorator({self._component.operation()})"

class DecoratorA (Decorator) :
    def operation (self) :
        return f"DecoratorA({self._component.operation()})"

class DecoratorB (Decorator) :
    def operation (self) :
        return f"DecoratorB({self._component.operation()})"
```

```
if (__name__ == "__main__") :

    component = Component()

    decorated = Decorator(component)
    print(decorated.operation())

    decorated_A = DecoratorA(component)
    print(decorated_A.operation())

    decorated_B = DecoratorB(decorated_A)
    print(decorated_B.operation())
```

[Result]
Decorator(Component)
DecoratorA(Component)
DecoratorB(DecoratorA(Component))

Observer

: 객체 상태가 변하는지를 감시하는 객체 존재
→ 여러 개의 객체가 특정 객체의 상태 변화를 감시/감지

```
class Subject :
    def __init__ (self) :
        self._observers = []

    def attach (self, observer) :
        self._observers.append(observer)

    def notify (self, message) :
        for observer in self._observers :
            observer.update(message)

class Observer :
    def update (self, message) :
        print(f"Observer notified : {message}")
```

```
if (__name__ == "__main__") :

    subject = Subject ()
    o1 = Observer()
    o2 = Observer()

    subject.attach(o1)
    subject.attach(o2)

    subject.notify("Interrupted")
```

[Result]
Observer notified : Interrupted
Observer notified : Interrupted

Strategy

: 사전에 strategy를 정의해놓고, 필요에 따라 전략 변경
→ 한 객체에 대해 수행방식을 변경가능

```
class Strategy :
    def execute (self) :
        pass

class StrategyA (Strategy) :
    def execute (self) :
        return "Strategy A"

class StrategyB (Strategy) :
    def execute (self) :
        return "Strategy B"
```

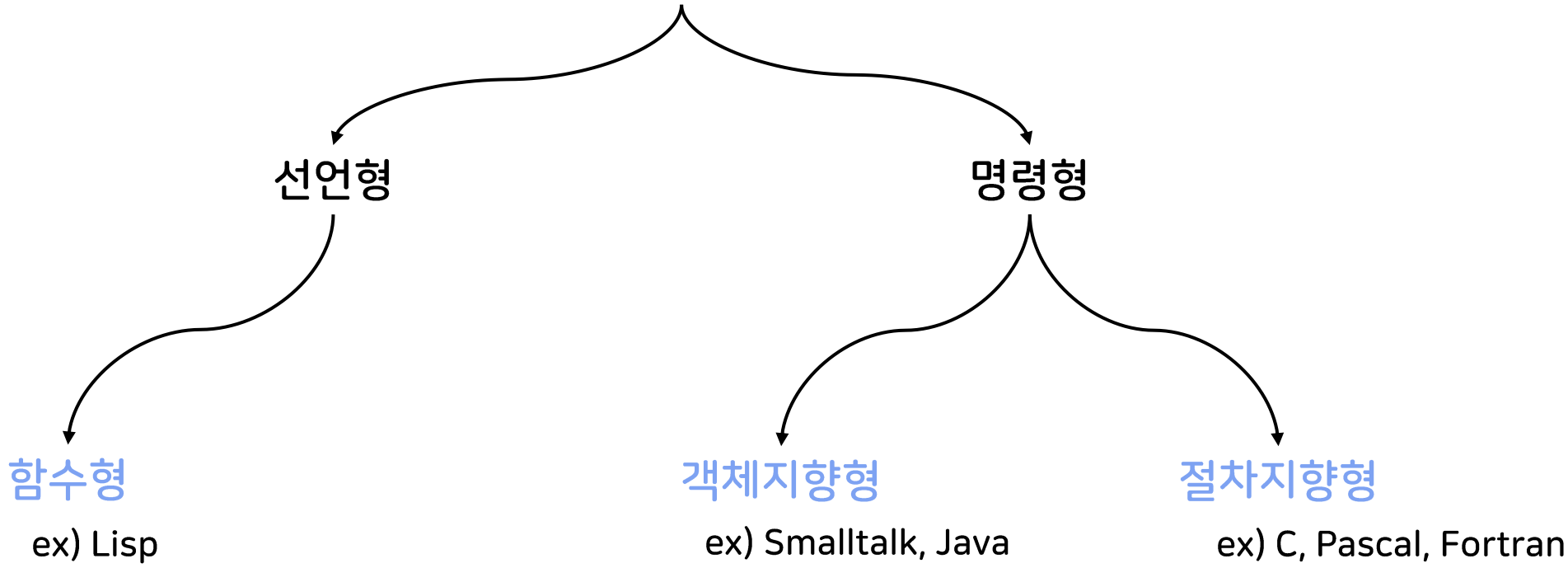
```
class Context :
    def __init__ (self, strategy) :
        self._strategy = strategy

    def set (self, strategy) :
        self._strategy = strategy

    def execute (self) :
        return self._strategy.execute()
```

```
if (__name__ == "__main__") :

    context = Context(StrategyA())
    print(context.execute())

    context.set(StrategyB())
    print(context.execute())
```

[Result]
Strategy A
Strategy B

프로그래밍 패러다임

프로그래밍 패러다임


선언형
명령형
함수형
ex) Lisp
객체지향형
ex) Smalltalk, Java
절차지향형
ex) C, Pascal, Fortran

Object-Oriented Programming (OOP)

객체지향형 언어

Encapsulation : 캡슐화. 데이터와 메서드를 묶어 내부 상태를 보호

Inheritance : 상속. 기존 클래스를 유지하고 새로운 클래스에서 재사용

Polymorphism : 다형성. 동일 이름의 메서드가 다르게 동작

Abstraction : 추상화. 객체의 세부 구현/동작과정은 감추고 사용 방법만 제공

Library

기능을 수행하기 위한 코드 모음
→ 개발자가 필요할 때 호출

개발자 중심

FrameWork

App의 구조를 미리 설계해 둠
→ 개발자는 규칙 내에서 작성

프레임워크 중심

보안

보안

for 데이터 기밀성/무결성, 서비스 가용성, 경제적 손실 방지, 개인 정보 보호, …

보호(Protection) : 내부 사용자나 프로그램 동작 과정에서 오류

보안(Security) : 외부 공격자나 악성 위협

Cryptography

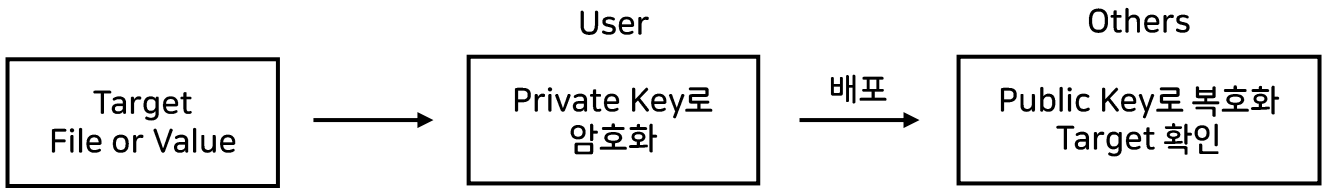
암호학

: 데이터의 기밀성, 무결성 등을 보장하기 위해 암호화/복호화 수행

- 대칭 키 암호화 : 암호화 key == 복호화 key. (AES, DES, …)

- 비대칭 키 암호화 : 암호화 key != 복호화 key. (RSA, ECC, …)

Private Key(개인만 알고있음)와 Public Key(만연하게 알려져있음)로 존재.
Private Key로 암호화? → Public Key로 복호화
Public Key로 암호화? → Private Key로 복호화



- Hashing : 데이터 무결성을 위해 사용. 암호화 목적은 아니며, 일방향 함수 (SHA-256, MD5, …)

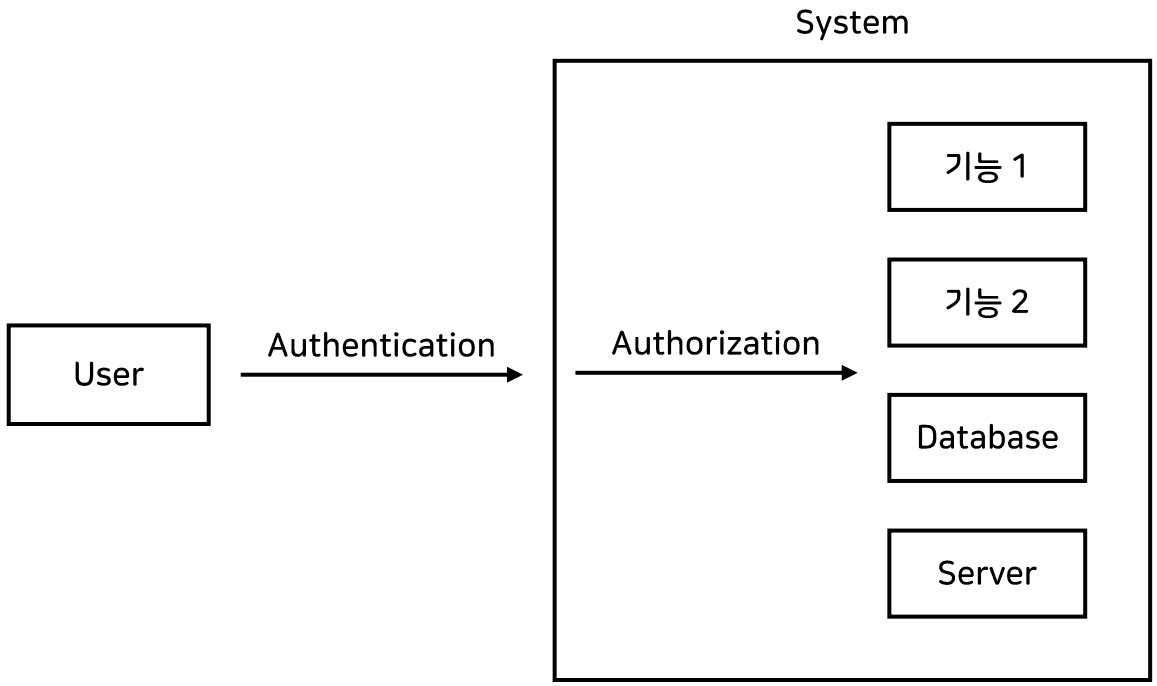
```
gaeng02@DESKTOP-C81RDG2:/bin$ md5sum python3
3f8ab1cc4780c2b2a73f257df651d163  python3
gaeng02@DESKTOP-C81RDG2:/bin$ sha1sum python3
482539628ffe2af8d64be2e0fe5b442464610650  python3
```

프로그램 설치 과정에서 변형이 발생할 수 있으므로,
기존 파일의 Hash 값과 설치 후 파일의 Hash 값을 비교하여 파일의 변형을 감지함
암호화의 목적이 아니고, 파일의 변형을 확인하기 위함

Authentication & Authorization

인증과 권한 부여

- Authentication (인증) : 사용자가 누구인지 확인하는 과정 (로그인, OTP, …)
- Authorization (권한부여) : 인증된 사용자가 어떠한 리소스에 접근할 수 있는지 결정 (OAuth, aws IAM)



SQL injection

: 사용자의 입력을 그대로 SQL 질의에 넣어 발생하는 보안 취약점
→ 데이터베이스를 조작하거나 민감한 정보 탈취

[SQL]

SELECT * FROM users WHERE username = “user_input” AND password = “password_input”;

[Input]

user_input = ‘ or ‘1’ = ‘1 # SQL에서 ‘는 주석이다. 뒷 문장이 다 주석처리가 되어 항상 참이 되도록 함.
password_input = {anything}

- Classic Injection : 위의 방식 대로 인증 우회
- Blind SQL Injection : 데이터를 직접 보지 못하는 상태에서 참/거짓에 따른 행동 차이 분석
- Union-based Injection : UNION 키워드를 추가해 데이터 노출
  (UNION : 조건절을 추가하여 데이터베이스 정보도 출력하도록 함. UNION SELECT *)
- Error-based Injection : 오류 메시지를 통해 데이터베이스 정보를 출력

Attack and Defense

BOF (Buffer OverFlow) : 메모리에서 Date 영역을 벗어나서 Code 영역까지 침범한 후, 악의적인 코드 삽입하는 공격기법

```
#include <iostream>
int main () {
   int array[5];
   for (int i = 0; i <= 5; i++)
      std::cin >> array[i];
}
```

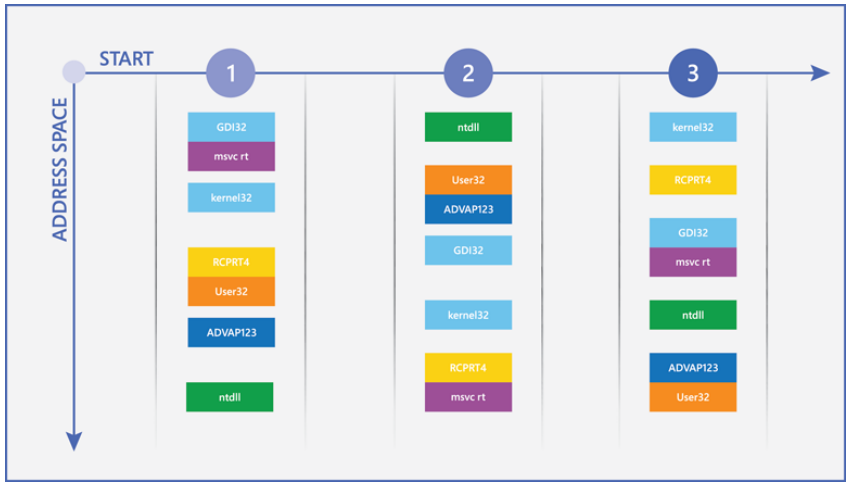
array size = 5
for 문에서 접근하는 횟수는 6이므로, over 접근
→ 이러한 것을 Buffer Overflow라 한다.

NX/DEP (No eXecute/Data Execution Prevention) : 메모리 상의 권한(Read/Write/Execute)을 이용함.
데이터로 들어온 경우 HW/OS 차원에서 실행 방지하는 보호기법

ROP (Return Oriented Programming) : 이미 존재하는 명령어로 이동함. 그 조각(Gadget)을 모아 실행시키는 공격기법

이미 기존에 존재하는 코드를 이용해서
공격자는 각 코드를 계속 이동하면서 한 조각(Gadget)씩 가져오고 이를 stack에 모아 실행시킴

ASLR (Address Space Layout Randomization) : 매번 함수를 호출할 때, 정해진 위치에서 호출하지 않고, 랜덤한 위치로 이동
Target 주소를 알 수 없도록 하는 보호기법



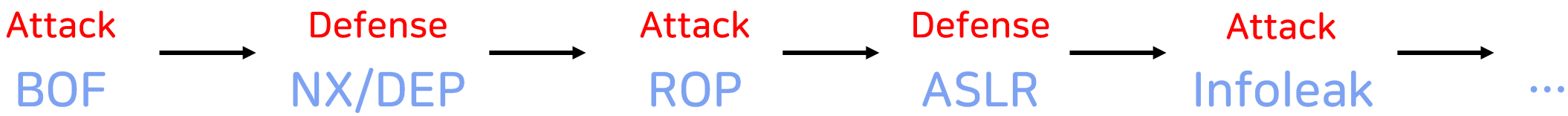
Microsoft Windows 10

매번 동작을 수행할 때 마다 호출되는 위치가 다름.
고정된 주소의 경우 공격자가 ROP 기법을 사용하여 공격할 수 있으나,
매번 랜덤된 주소로 변경되므로 ROP 공격을 사용할 수 없음.

Infoleak (Information Leak) : 메모리의 내용을 노출시키는 버그. BOF와 의도치 않은 버그를 이용한 공격기법

```
printf("%s");
```

%s는 forma으로, 뒤에 출력할 인자가 필요하지만 현재 없음
따라서 의도치 않게 memory의 일부가 출력되는 버그가 발생하여
이를 이용해 공격자가 공격을 시도할 수 있음.


Attack
BOF
Defense
NX/DEP
Attack
ROP
Defense
ASLR
Attack
Infoleak
…

Task deadline : 11/26 (화))

1. [단답형] 게임을 만들 때, 하나의 캐릭터(main)를 두고 이 캐릭터에 특정 장비나 아이템을 장착, 해제 한다.
   이 상황에서 사용할 수 있는 디자인 패턴은 무엇인가?

2. [서술형] React Native는 라이브러리일까, 프레임워크일까?

3. [서술형] SQL injection을 막을 수 있는 방법은 무엇인가?

Curriculum

1주차 : Orientation	(9/4)
2주차 : Data Structure	(9/11)
3주차 : Algorithm 1	(9/25)
4주차 : Algorithm 2	(10/2)
5주차 : Computer Architecture	(10/16)
6주차 : Operating System	(11/6)
7주차 : Computer Network	(11/13)
8주차 : DataBase	(11/20)
9주차 : Design Pattern & Security	(11/27)
10주차 : Real Problems	(12/4)
11주차 : Real Problems Feedback	(12/11)