

시스템 보안

#4 80x86 시스템 - 실습 (디버거)



Python 설치 (Windows 기준)

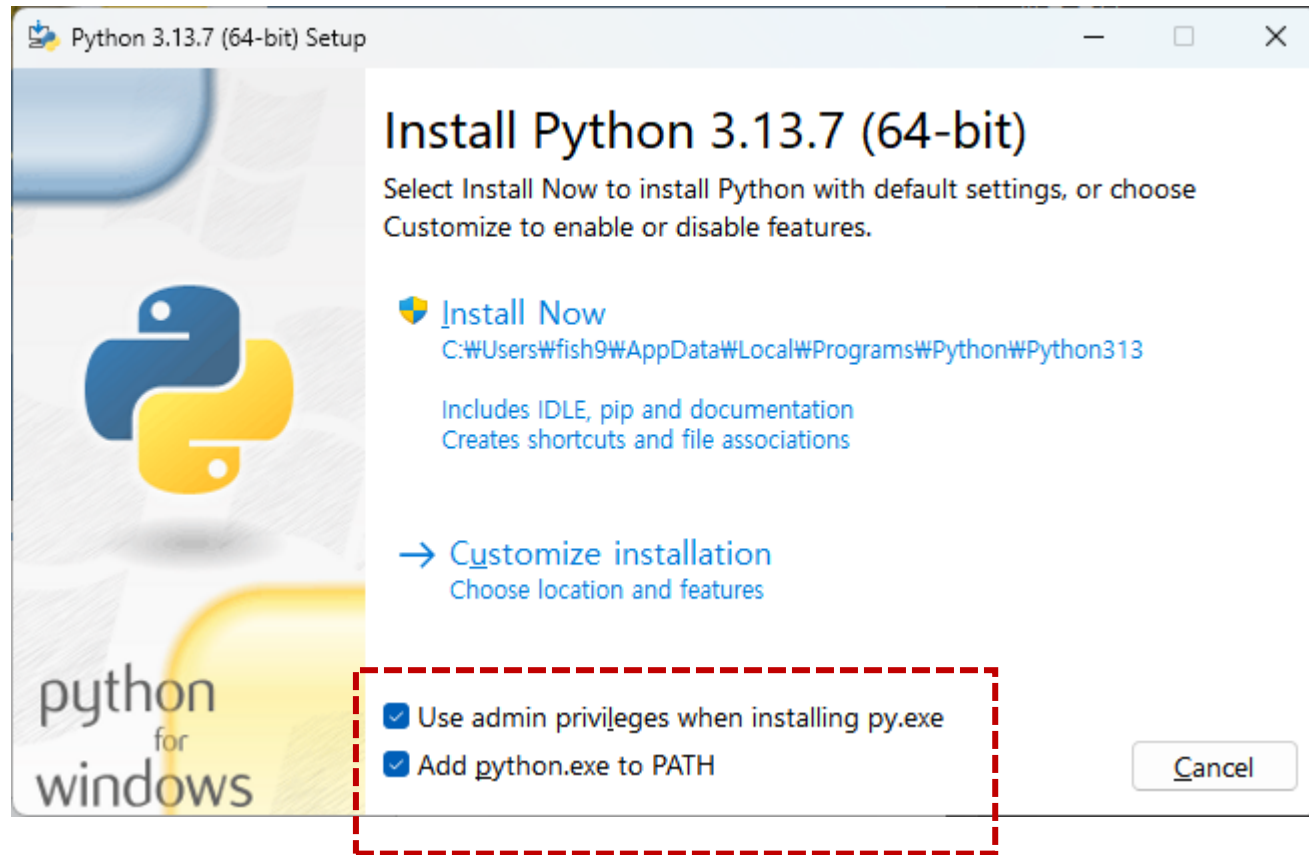
- Python 설치

<https://www.python.org/downloads/>



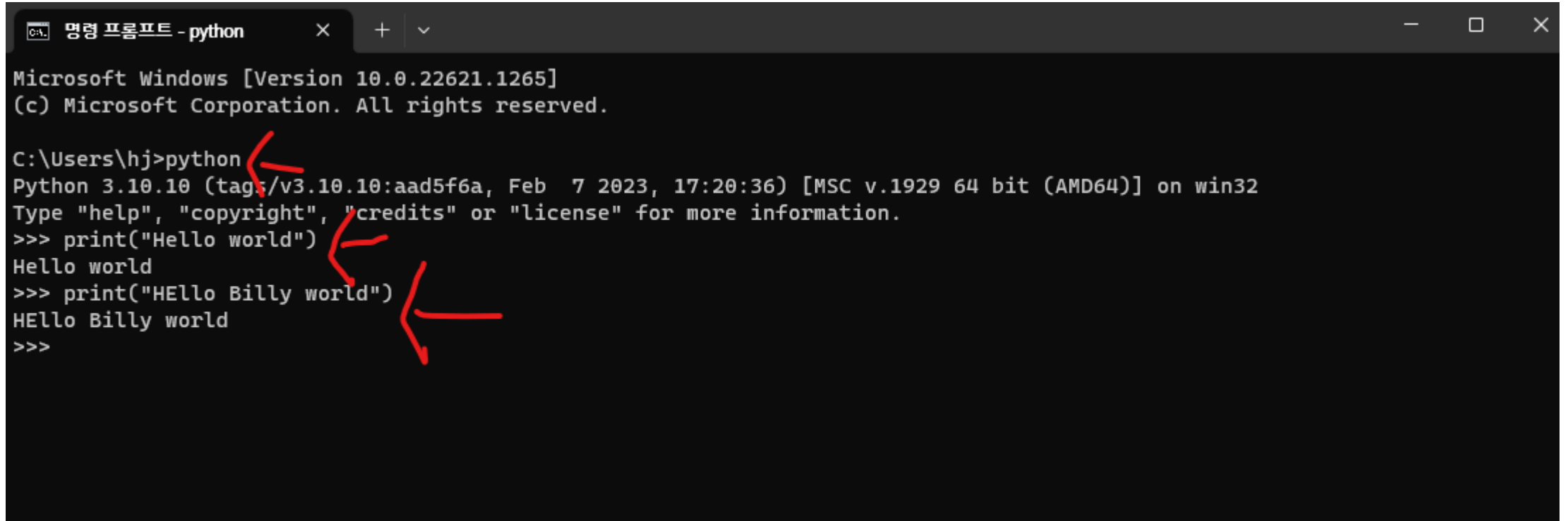
- 홈페이지에서 설치 파일 다운로드

- Python 설치



- 옵션 체크, 체크 하지 않아도 설치 가능하지만 단독으로 사용하는 경우 체크하는 게 편함
- 각자 사용하는 파이썬 개발 환경이 있으면 그대로 사용하면 됨 (별도 설치 불필요)

- Python 설치



```
명령 프롬프트 - python
Microsoft Windows [Version 10.0.22621.1265]
(c) Microsoft Corporation. All rights reserved.

C:\Users\hj>python
Python 3.10.10 (tags/v3.10.10:aad5f6a, Feb  7 2023, 17:20:36) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world")
Hello world
>>> print("Hello Billy world")
Hello Billy world
>>>
```

- 정상 설치 테스트

Python 설치 (Windows 기준)

- 가상 환경 설정 (virtual environment)

- **venv : python에서 기본으로 제공하는 표준 라이브러리로 별도 설치 불필요**

① **cd [환경을 만들고자 하는 경로]**

② **create**
python -m venv my_env

③ **activate**
.\my_env\Scripts\activate.bat

④ **delete**
my_env 경로 삭제

⑤ **activate 상태에서는 가상환경 내에 패키지 설치 제거가 이루어짐**

Breakpoint 설정 방법

x86 시스템의 구조

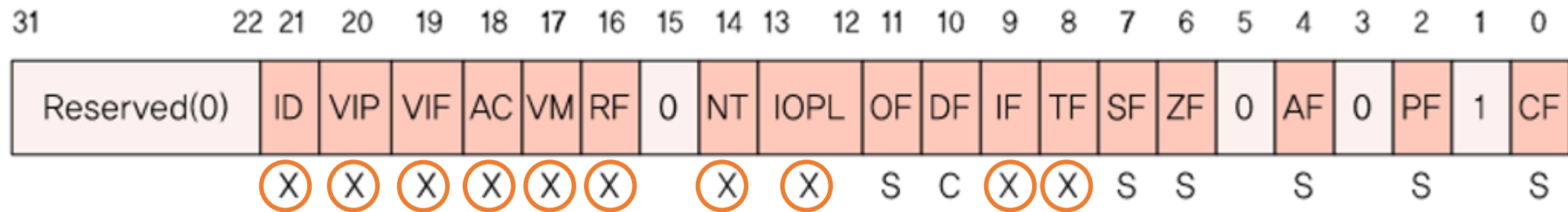
- S/W BP(Break Point)와 H/W BP 비교
 - S/W BP는 소프트웨어 인터셉트(INT3)를 통해 코드 실행 지점에서 중단점을 설정
 - H/W BP는 CPU의 하드웨어 레지스터를 사용해 메모리 접근 또는 명령어 실행을 감시
 - S/W BP는 유연하지만 성능 저하와 코드 변조 가능성이 존재
 - H/W BP는 빠르고 정확하며 코드 무결성을 유지
 - S/W BP는 디버거에서 쉽게 설정 가능하나 HW BP는 제한된 개수만 설정 가능

Trap Flag & Resume Flag

x86 시스템의 구조

- x86 시스템 구조 – CPU : 레지스터(Register) 종류와 기능

- 플래그 레지스터(Flag Register) : EFLAGS 레지스터
- 시스템 플래그(System Flag)



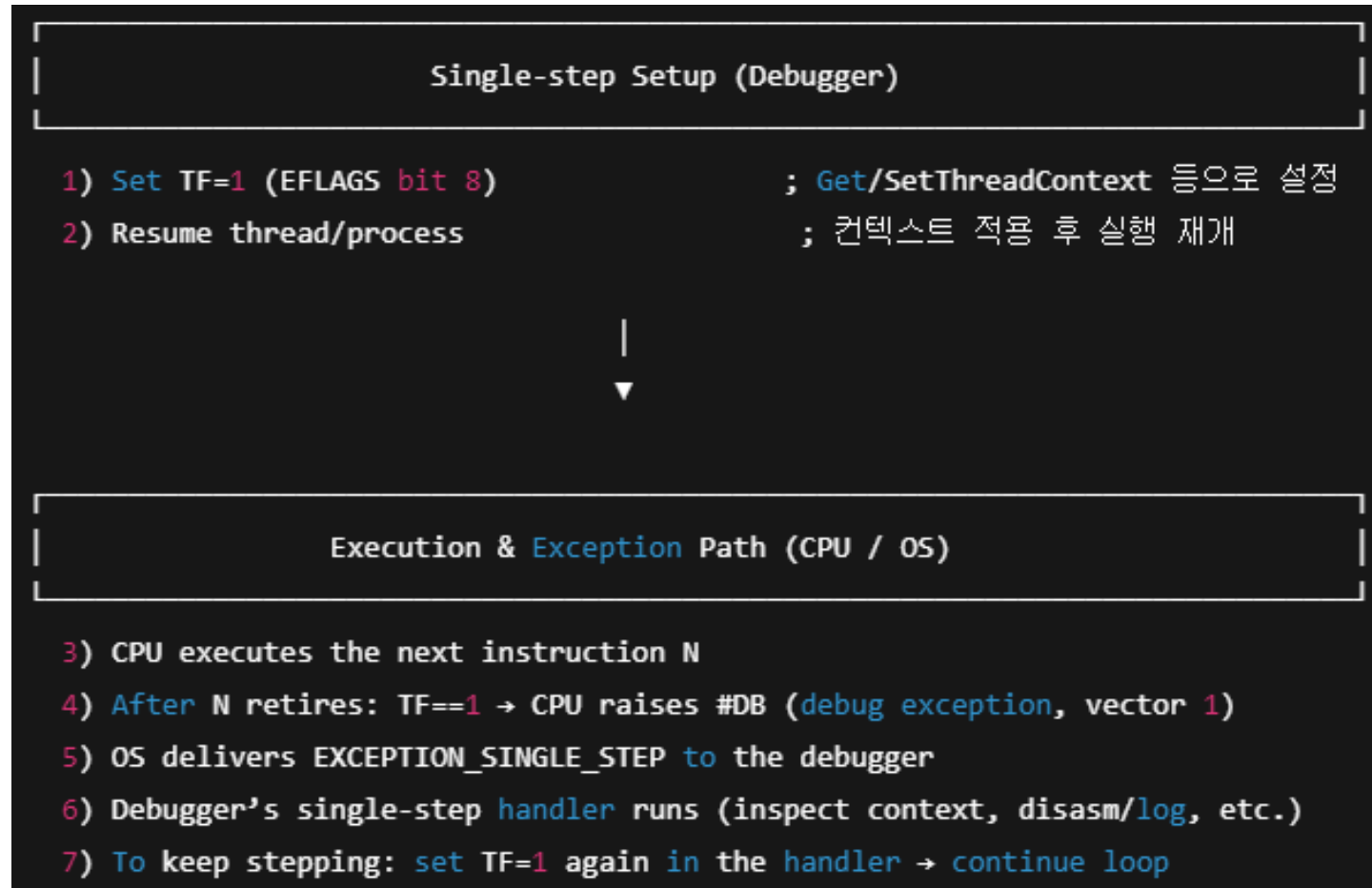
- TF(Trap Flag, 비트 8) : 디버깅 시 'Single Step Mode' 모드를 활성화하면 세트
- RF(Resume Flag, 비트 16) : 프로세서의 디버그 예외 반응을 제어, 1로 설정 시 디버그 오류를 무시하고 다음 명령어를 수행

x86 시스템의 구조

- x86 시스템 구조 – CPU : 레지스터(Register) 종류와 기능
 - 플래그 레지스터(Flag Register) : EFLAGS 레지스터
 - TF(Trap Flag) – Thread 별 적용
- Trap Flag는 x86 프로세서의 디버깅에 사용되는 비트
- TF는 프로세서 상태 레지스터의 8번째 비트에 위치
- TF가 설정되면 한 명령어 실행 후 인터럽트가 발생
- 디버거는 TF를 통해 단계별 실행(step-by-step)을 구현
- TF는 소프트웨어 디버깅과 문제 해결에 중요한 역할을 함

x86 시스템의 구조

- x86 시스템 구조 - CPU : 레지스터(Register) 종류와 기능



- TF는 주로 Single Step 디버깅 처리를 위해 사용

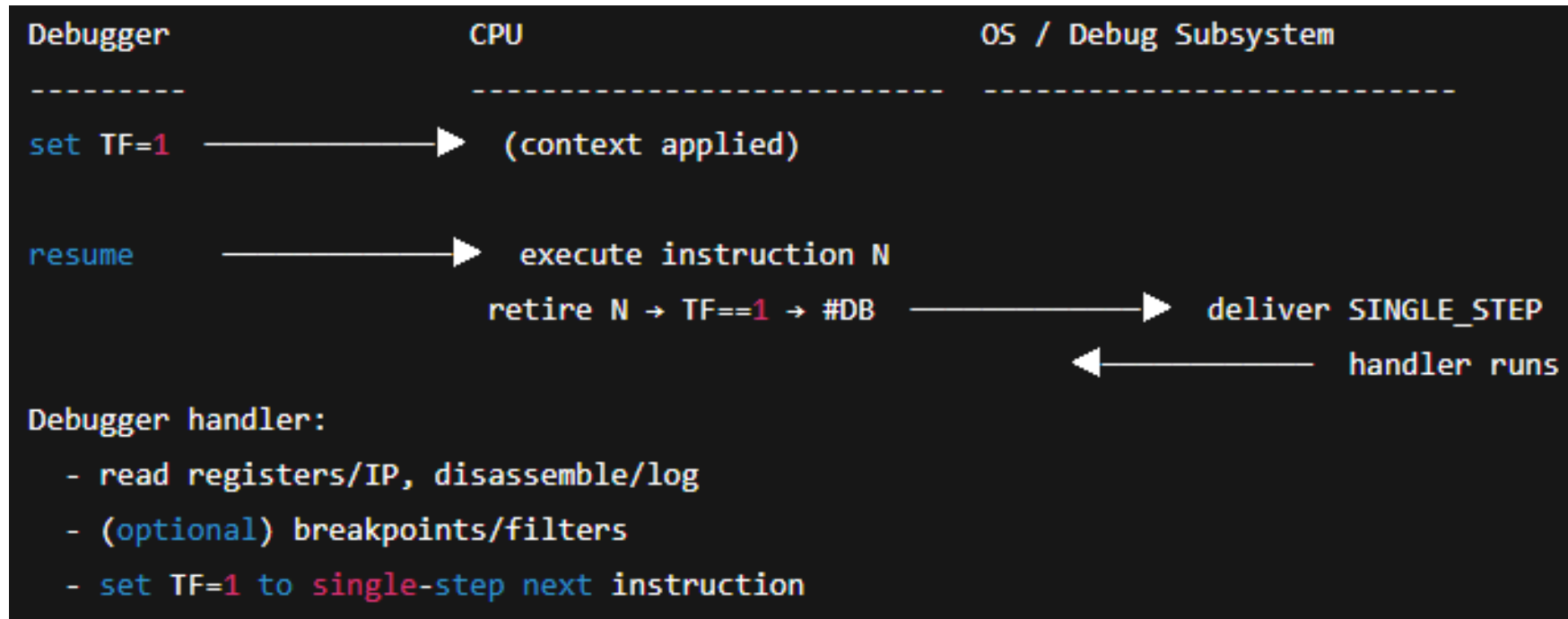
- EFLAGS 레지스터의 TF Flag를 이용

- TF 설정 시 하나의 인스트럭션 실행 후 Handler 루틴이 호출됨

- 다음 인스트럭션도 계속 디버깅 하려면 TF 재설정

x86 시스템의 구조

- x86 시스템 구조 - CPU : 레지스터(Register) 종류와 기능



x86 시스템의 구조

- x86 시스템 구조 – CPU : 레지스터(Register) 종류와 기능
 - 플래그 레지스터(Flag Register) : EFLAGS 레지스터
 - RF(Resume Flag)
- RF는 x86 프로세서 상태 레지스터의 16번째 비트
- 디버깅 중에 특정 인터럽트나 예외 발생을 일시적으로 무시할 때 사용
- RF가 설정되면 디버거가 지정한 예외나 인터럽트를 재개
- 단계별 실행을 중단하고 재개하는 데 중요한 역할을 함
- RF는 Trap Flag와 함께 디버깅 제어를 지원

x86 시스템의 구조

- x86 시스템 구조 - CPU : 레지스터(Register) 종류와 기능

SW Breakpoint Handling (Debugger)

- 1) INT3 hit → #BP ; IP points AFTER the 0xCC byte
- 2) Restore original byte at BP address ; write back saved opcode byte
- 3) Set IP ← BP address ; rewind instruction pointer
- 4) Set RF=1 (EFLAGS bit 16) ; optionally: set TF=1 to single-step next



Execution & Exception Outcome (CPU / OS)

- 5) CPU executes the restored instruction
- 6) RF suppresses #DB for exactly THIS one instruction
- 7) RF auto-clears (RF=0 again)
- 8) If TF==1 → CPU raises #DB after retire → OS delivers EXCEPTION_SINGLE_STEP
- 9) Debugger's single-step handler runs (inspect, log, re-arm BP, etc.)

└ Loop: if you want to keep stepping, set TF=1 again in the handler and continue.

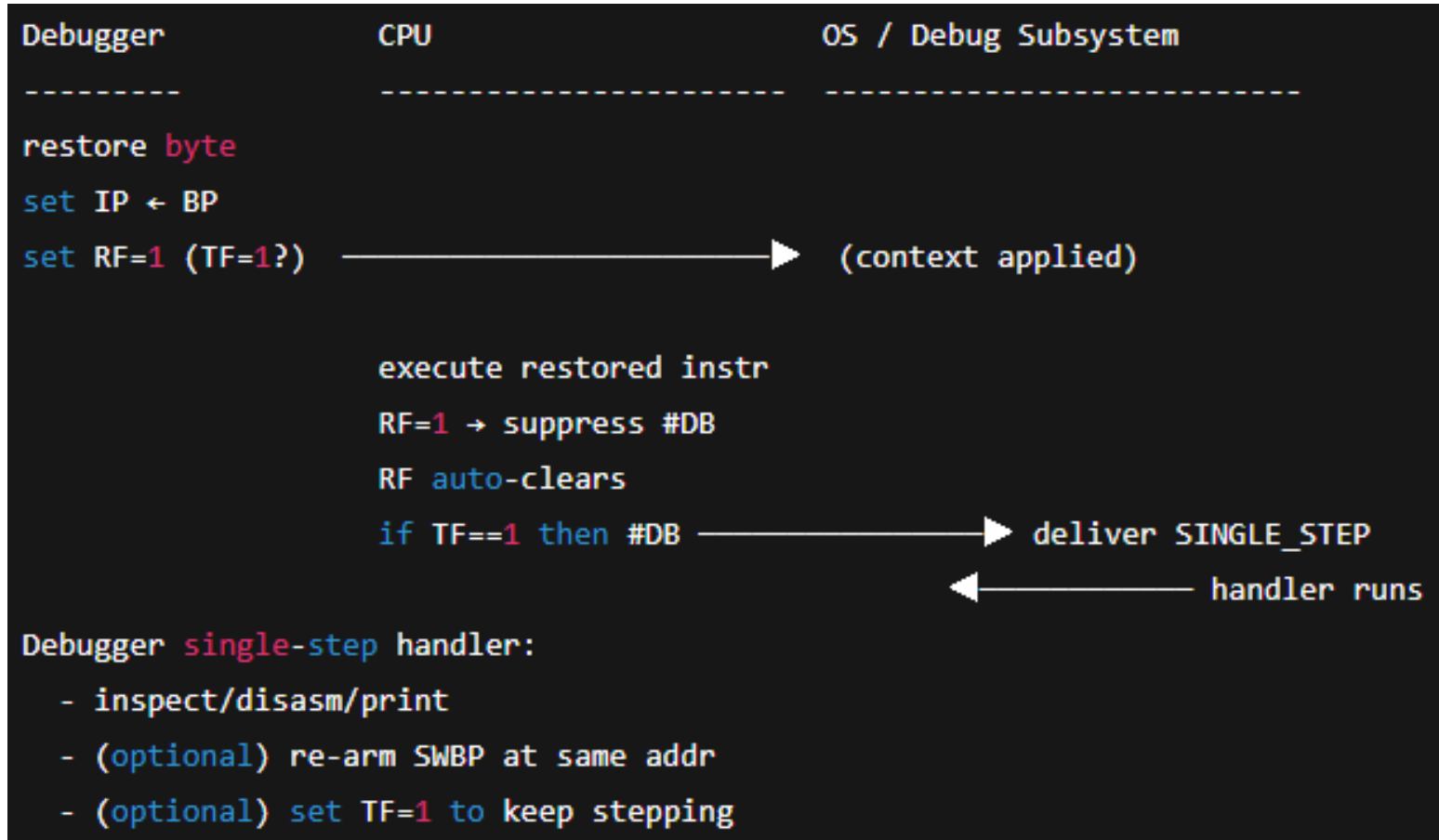
- RF는 주로 S/W BP 처리를 위해 사용

- INT3으로 수정한 코드를 원래 코드로 복원시키고, INT3 다음 위치로 이동한 IP를 원래 위치로 이동 (-1)

- IP가 원래 코드 위치로 이동한 상태에서 코드를 실행할 때 해당 코드에 대해서는 BP가 발생하지 않도록 RF 설정

x86 시스템의 구조

- x86 시스템 구조 - CPU : 레지스터(Register) 종류와 기능



디버거 작동 방식 실습

(singlestep.py)

- WinAppDbg (Windows 전용 Python 라이브러리)

- Windows Debug API를 Python에서 다루게 해주는 고수준 래퍼 라이브러리
- 디버거를 직접 만들지 않고도 프로세스 실행/Attach, 단일 스텝, 브레이크포인트, 메모리 읽기/쓰기, 스레드/모듈 정보 등을 손쉽게 자동화
- 설치 : `pip install git+https://github.com/MarioVilas/winappdbg.git@master`

- ✓ Key features

- 프로세스/스레드 제어: `execv`, `attach`, `suspend/resume`, 컨텍스트(EIP/RIP, EFLAGS) 읽기/쓰기
- 예외/이벤트 처리: `EXCEPTION_SINGLE_STEP`, `EXCEPTION_BREAKPOINT` 등 핸들러 기반
- 메모리/모듈: 원격 메모리 `read/write`, 이미지 베이스·모듈 나열, 간단한 PE 정보
- S/W 및 H/W 브레이크포인트 관리

- WinAppDbg (Windows 전용 Python 라이브러리)

```
from winappdbg.debug import Debug
from winappdbg.event import EventHandler
from winappdbg import win32 # DBG_CONTINUE 등 상수

class H(EventHandler):
    def exception(self, event):
        code = event.get_exception_code()
        print(f"exception: 0x{code:08X}")
        event.continueStatus = win32.DBG_CONTINUE # 0x00010002와 동일

dbg = Debug(H())
try:
    dbg.execv(["notepad.exe"])
    dbg.loop()
finally:
    dbg.stop()
```

- Capstone (멀티 아키텍처 디스어셈블러)

- C 기반 고성능 디스어셈블 엔진 + 파이썬 바인딩
- 바이너리 코드에서 바이트 → 명령어(니모닉 등)로 빠르고 안정적으로 변환
- 설치 : `pip install capstone`

- ✓ Key features

- 순수 디스어셈블(에뮬레이션/역공학 IR 변환은 아님)
- 스트리밍/슬라이딩 윈도우 방식으로 빠른 처리, 명령 바이트 길이 제공
- 관대한 라이선스(BSD)

- Capstone (멀티 아키텍처 디스어셈블러)

```
from capstone import Cs, CS_ARCH_X86, CS_MODE_64
md = Cs(CS_ARCH_X86, CS_MODE_64)
code = b"\x55\x48\x8B\xEC\x48\x83\xEC\x20" # 예: function prologue
for ins in md.disasm(code, 0x140001000):
    print(f"0x{ins.address:X}: {ins.mnemonic} {ins.op_str} ; len={len(ins.bytes)}")
```

QA

