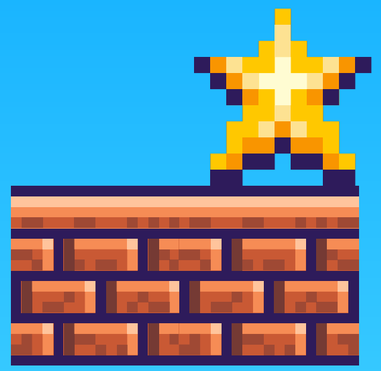
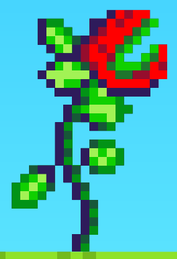
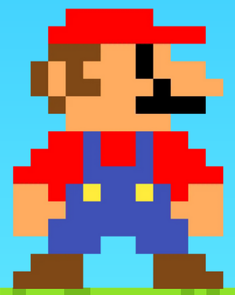
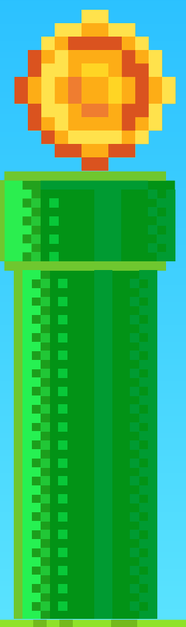


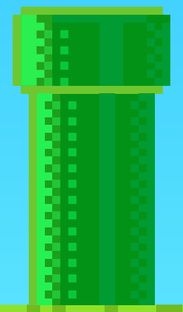
INTERESTING ASSEMBLY INSTRUCTIONS

닌텐도 구조대









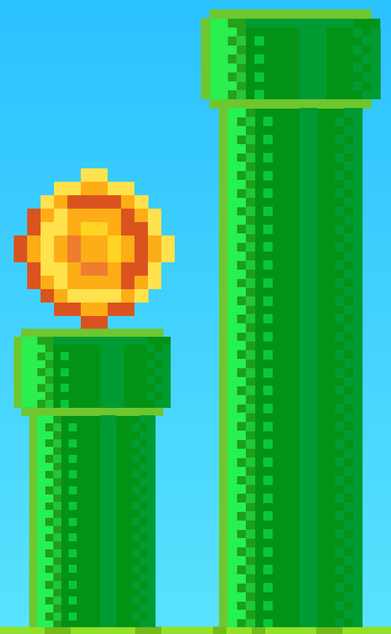
HOW TO PLAY

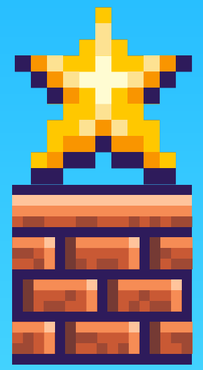
There are 3 different types of games!

kernal ASM

VMX ASM

AES-NI






ARE YOU
READY?

서론

신기한 어셈블리 명령어(Interesting Assembly Instructions)

- CPU가 제공하는 특수 목적 명령어들
- 커널·VM·암호화 등 다양한 영역에서 활용
- 하드웨어 수준 최적화 이해

kernel ASM

- 커널의 역할?
- 시스템 권한(Ring0)에서 하드웨어 접근을 독점한다.
  유저 프로그램은 커널의 인터페이스(system call)를 통해서만 하드웨어에 접근한다.

- 커널에서만 사용되는 특별한 명령어..?
  - HLT
  - swapgs
  - iretq , sysret
  - wrmsr , rdmsr
  - cli / sti

  등등...

kernel ASM

- HLT F4

- CPU 실행을 중단하고 "대기 상태"로 들어간다.
- 인터럽트, NMI, Reset 등의 이벤트가 발생해야 다시 실행 재개됨.
- Ring0 전용
  → 유저모드에서 실행하면 #GP Fault 발생.
- 전력 절약 good.

- STI FB

- RFLAGS 레지스터의 IF(Interrupt Flag) bit = 1
- 외부 하드웨어 인터럽트가 다시 들어올 수 있음

- CLI FA

- IF를 0으로 클리어 → IRQ 비활성화

kernel ASM

```
; Attributes: noreturn

                public default_idle
default_idle    proc near               ; CODE XREF: amd_e400_idle+1C↑p
                                        ; amd_e400_idle:loc_FFFFFFFF8103A598↑j
                                        ; DATA XREF: ...
                endbr64
                jmp     short loc_FFFFFFFF81E07ECD
; ---------------------------------------------------------------------------
                dw 0Fh
; ---------------------------------------------------------------------------
                sub     eax, 3FF063h

loc_FFFFFFFF81E07ECD:                   ; CODE XREF: default_idle+4↑j
                sti
                hlt
default_idle    endp
```

- LIDT 0F 01 /2

- IDTR(Interrupt Descriptor Table Register)을 새로운 IDT로 설정

* 프로세서에게 인터럽트가 걸렸을 경우에 수행해야 할 핸들러들의 벡터와 벡터의 정보 등을 저장해 두는 구조체가 IDT(Interrupt Descriptor Table)
* IDTR은 시스템의 IDT 가 존재하는 메모리 주소의 베이스 어드레스와 IDT entry 의 갯수가 저장되기로 약속된 레지스터

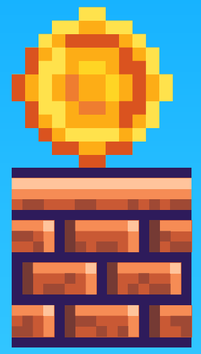
```
; __int64 idt_setup_early_traps(void)
                public idt_setup_early_traps
idt_setup_early_traps proc near         ; CODE XREF: setup_arch+48↓p
                endbr64
                mov     edx, 1
                mov     esi, 2
                mov     rdi, offset early_idts
                call    idt_setup_from_table_constprop_0
                lidt    fword ptr cs:idt_descr
                jmp     __x86_return_thunk
idt_setup_early_traps endp
```

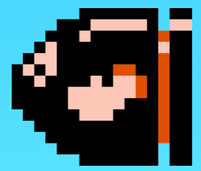
kernel ASM

- RDMSR(Read From Model Specific Register) 0F 32
- MSR 레지스터를 읽음 (EDX:EAX)

- WRMSR (Write to Model Specific Register) 0F 30
- MSR 레지스터에 씀

- MSR = Model-specific register 모델별 레지스터. CPUID 명령어를 통해 확인 가능





kernel ASM

- SWAPGS — 0F 01 F8
- 유저 GS_base ↔ 커널 GS_base 교환

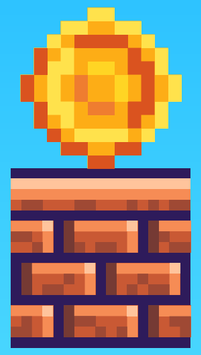
- IRETQ — 48 CF

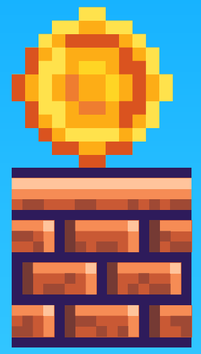
Ring0 → Ring3 복귀 시 쓰이는 핵심 명령어

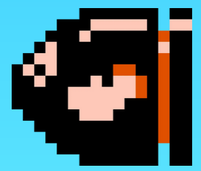
swapgs_restore_regs_and_return_to_usermode() 참조

```
void ret2shell(){

 commit_creds(prepare_kernel_cred(0));
     asm("swapgs;"
         "mov %%rsp, %0;"
         "iretq;"
         : : "r" (&tf));
    return;
}
```







VMX ASM

- VMX란?
(Virtual Machine Extensions)

- Intel CPU의 하드웨어 가상화 확장
- Intel이 제공하는 VT-x(Virtualization Technology) 의 핵심 구성 요소

- CPU가 "가상 머신(Guest)"을 하드웨어 차원에서 직접 실행할 수 있도록 도와주는 기능

- 기존에는 소프트웨어 가상화에 의해 느렸던 동작들을 CPU가 직접 처리하여 성능을 크게 향상시킴

VMX Instructions

Mnemonic	Summary
INVEPT	Invalidate Translations Derived from EPT
INVVPID	Invalidate Translations Based on VPID
VMCALL	Call to VM Monitor
VMCLEAR	Clear Virtual-Machine Control Structure
VMFUNC	Invoke VM function
VMLAUNCH	Launch/Resume Virtual Machine
VMPTRLD	Load Pointer to Virtual-Machine Control Structure
VMPTRST	Store Pointer to Virtual-Machine Control Structure
VMREAD	Read Field from Virtual-Machine Control Structure
VMRESUME	Launch/Resume Virtual Machine
VMRESUME (1)	Resume Virtual Machine
VMWRITE	Write Field to Virtual-Machine Control Structure
VMXOFF	Leave VMX Operation
VMXON	Enter VMX Operation

VMX ASM

- VMXON F3 0F C7 31
- CPU에서 VMX operation을 활성화
- 즉, VT-x 모드를 켜는 동작
- 한 프로세서(logical CPU)에서 VMX를 켜면 그 코어는 하이퍼바이저(Host/Root)·게스트(Non-root)를 관리할 수 있는 상태가 된다.
- VMXOFF 0F 01 C4
- VMX 모드를 비활성화
- 동작흐름

  유저는 커널권한으로 이동 → CPL=0 확보
  cr4의 VMXE 비트를 설정한다.
  vmxon [vmxon_region_phys] 명령 실행.
  CPU 내부 상태가 VMX operation으로 전환되고, 해당 logical processor는 VMX 명령어를 수행할 수 있게 된다.
  VMXON 성공 후에는 VM명령어들 사용 가능.
  VM 끝내려면 vmxoff로 VMX operation 해제.

VMX ASM

- VMPTRLD vmptrld m64 — VMCS region(4KB)의 주소.
- (Load Pointer to Virtual-Machine Control Structure) 0F C7 33
- VMCS pointer(현재 사용할 VMCS) 로드
  4KB VMCS structure 존재해야 함
  그 물리주소를 operand로 넣어서 로드
  CPU가 internal VMCS pointer를 갱신
- 가상화 기술에서 하이퍼바이저가 호스트와 게스트(가상 머신) CPU 상태를 전환하고 관리하기 위해 사용하는 핵심 데이터 구조

- VMCS(Virtual Machine Control Structure)란 무엇인가?

- VMCS는 CPU가 가상 머신(Guest)의 동작을 관리하기 위해 사용하는 4KB 크기의 하드웨어 제어 구조이다.

→ 하이퍼바이저가 VM을 통제하는 모든 규칙·상태·레지스터 값이 들어 있는 하드웨어 구조.

→ VMCS Header (revision id 포함) , VMCS Abort Indicator , VMCS Data Fields

VMX ASM

- VMPTRST VMPTRST m64 -저장할 메모리 위치.
- (Store Pointer to Virtual-Machine Control Structure)
- 0F C7 3B
- 현재 VMCS 포인터를 지정된 메모리 주소에 저장합니다.

- VMWRITE VMWRITE r64, r/m64 - 64비트 레지스터(값) , register OR memory(필드)
- (Write Field to Virtual-Machine Control Structure)
- NP 0F 79
- VMX 루트 동작 시에는 현재 VMCS에 기록하고, VMX 비루트 동작 시에는 현재 VMCS의 • VMCS 링크 포인터 필드가 참조하는 VMCS에 기록합니다.

- VMREAD VMREAD r/m64, r64 - 읽고 싶은 VMCS 필드, 읽은 값을 저장할 레지스터
- (Read Field from Virtual-Machine Control Structure)
- NP 0F 78
- 지정된 VMCS 필드값을 읽어 레지스터로 반환.

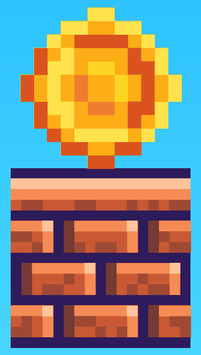
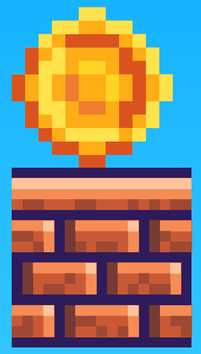
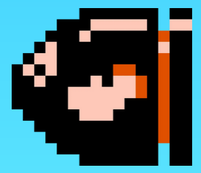
VMX ASM

- VMLAUNCH/VMRESUME — Launch/Resume Virtual Machine
- 0F 01 C2 / 0F 01 C3
- 현재 로드된 VMCS를 기반으로 게스트를 최초 진입시킨다.
- guest를 재실행한다.
- VMCALL — Call to VM Monitor
- 0F 01 C1
- guest에서 실행하면 VM Exit(=VMCALL exit)으로 하이퍼바이저에게 제어를 넘기는 역할.
- VMFUNC — Invoke VM Function
- 0F 01 D4
- 이 명령어를 사용하면 VMX 비루트 동작 영역의 소프트웨어가 VM 기능을 호출할 수 있습니다. VM 기능은 VMX 루트 동작 영역의 소프트웨어에서 활성화 및 구성된 프로세서 기능입니다. EAX 값은 호출할 특정 VM 기능을 선택합니다

VMX ASM

- VMCLEAR — Clear Virtual-Machine Control Structure
- VMCLEAR m64
- 66 0F C7
- 이 명령어는 해당 VMCS의 데이터를 메모리의 VMCS 영역으로 복사합니다. 또한 VMCS 영역의 일부를 초기화
  더 이상 사용 불가하게 표시.

- VM 명령어들은 Type-1 또는 Type-2 하이퍼바이저에서 사용됨
  ex) KVM (Linux Kernel Virtual Machine), Hyper-V (Microsoft)

AES

- AES (Advanced Encryption Standard)란?
- 현대 암호 시스템에서 가장 널리 사용되는 대칭키 블록 암호 표준

- 핵심 특징

- 대칭키 암호
- 블록 크기: 128비트 고정
- 키 길이: 128 / 192 / 256bit
- 라운드 구조 기반 암호

- AES-NI

- AES의 라운드 연산은 연산량이 많기 때문에
  CPU가 직접 라운드 단계를 명령어 하나로 처리하는 AES-NI가 등장

AES-NI

- **AES-NI (AES New Instructions)**
- Intel 및 AMD CPU가 제공하는 AES 전용 하드웨어 가속 명령어 세트

- **왜 AES-NI가 필요한가?**
- 기존 소프트웨어 AES는 느리고 복잡
- CPU가 AES 연산을 특수 회로로 구현하여 단일 명령어로 수행
- 일정한 실행 시간 확보

- **핵심 아이디어**
- SubBytes / ShiftRows / MixColumns / AddRoundKey 를 AESENC로 한 번에 처리
- InvSubBytes / InvShiftRows / InvMixColumns / AddRoundKey를 AESDEC로 한 번에 처리

- **실제 사용 예시**
- Chrome, Firefox, Edge + OpenSSL, BoringSSL, LibreSSL 등
- VPN / IPsec / WireGuard
- Linux dm-crypt, LUKS, Windows BitLocker, macOS FileVault

AES-NI

- AES-NI 핵심 명령어
- AESENC xmm, xmm
  - AESENC = SubBytes + ShiftRows + MixColumns + AddRoundKey
- AESENCLAST xmm, xmm
  - 마지막 라운드 암호화
- AESDEC xmm, xmm
  - AESDEC = InvSubBytes + InvShiftRows + InvMixColumns + AddRoundKey
- AESDECLAST xmm, xmm
  - 마지막 라운드 복호화
- AESIMC xmm, xmm
  - 복호화를 위한 reverse key 생성
- AESKEYGENASSIST xmm, imm8
  - 키 스케줄 보조 명령어

AES

▪ main 함수

```
main:
    ; 프로로그: RSP를 16바이트 정렬
    push    rbp             ; entry rsp%16=8 → 0
    mov     rbp, rsp
    sub     rsp, 16         ; 여유 스택, 여전히 0 정렬

    ; 0) AES-128 마스터 키 출력
    mov     rdi, fmt_key_hex
    xor     eax, eax
    call    printf

    lea     rdi, [master_key]
    call    print_block_hex

    mov     eax, [master_key]   ; 4바이트 로드 (00 01 02 03 → 0x03020100)
    movsx   rsi, eax            ; 정수 인자로 전달
    mov     rdi, fmt_key_int
    xor     eax, eax
    call    printf

    ; 1) 키 스케줄 생성 (AESKEYGENASSIST, AESIMC 사용)
    lea     rdi, [master_key]
    lea     rsi, [enc_round_keys]
    call    aes128_expand_key

    lea     rdi, [enc_round_keys]
    lea     rsi, [dec_round_keys]
    call    aes128_invert_keys
```

```
; 2) 정수 입력
mov     rdi, fmt_int_input
xor     eax, eax
call    printf

mov     rdi, fmt_scan_int
lea     rsi, [user_int]
xor     eax, eax
call    scanf

; 3) 평문 블록 구성 (앞 4바이트 = 입력 정수, 나머지 0)
lea     rdi, [plain_block]
mov     rcx, 16
xor     eax, eax
rep     stosb

mov     eax, [user_int]
mov     [plain_block], eax
```

AES

main 함수

```
; 4) 암호화
lea     rdi, [plain_block]
lea     rsi, [cipher_block]
lea     rdx, [enc_round_keys]
call    aes128_encrypt_block

mov     rdi, fmt_out_enc
xor     eax, eax
call    printf

lea     rdi, [cipher_block]
call    print_block_hex

; 5) 복호화
lea     rdi, [cipher_block]
lea     rsi, [decrypted_block]
lea     rdx, [dec_round_keys]
call    aes128_decrypt_block

mov     rdi, fmt_out_dec
xor     eax, eax
call    printf

lea     rdi, [decrypted_block]
call    print_block_hex
```

```
; 6) 복호 결과의 앞 4바이트 출력
mov     eax, [decrypted_block]
movsx   rsi, eax
mov     rdi, fmt_out_int
xor     eax, eax
call    printf

; 에필로그
add     rsp, 16
pop     rbp
mov     eax, 0
ret
```

AES

• aes128_expand_key 함수

```
aes128_expand_key:
    push    rbp
    mov     rbp, rsp

    movdqu  xmm0, [rdi]          ; master key
    movdqu  [rsi], xmm0          ; round key 0

    AES_128_KEYEXP 0x01, rsi+16
    AES_128_KEYEXP 0x02, rsi+32
    AES_128_KEYEXP 0x04, rsi+48
    AES_128_KEYEXP 0x08, rsi+64
    AES_128_KEYEXP 0x10, rsi+80
    AES_128_KEYEXP 0x20, rsi+96
    AES_128_KEYEXP 0x40, rsi+112
    AES_128_KEYEXP 0x80, rsi+128
    AES_128_KEYEXP 0x1B, rsi+144
    AES_128_KEYEXP 0x36, rsi+160

    pop     rbp
    ret
```

```
%macro AES_128_KEYEXP 2
    ; xmm0 : 이전 라운드 키
    ; %1   : rcon (imm8)
    ; %2   : 저장할 주소 (예: rsi+16)

    aeskeygenassist xmm1, xmm0, %1
    pshufd          xmm1, xmm1, 0xff

    movdqa  xmm2, xmm0
    pslldq  xmm2, 4
    pxor    xmm0, xmm2

    movdqa  xmm2, xmm0
    pslldq  xmm2, 4
    pxor    xmm0, xmm2

    movdqa  xmm2, xmm0
    pslldq  xmm2, 4
    pxor    xmm0, xmm2

    pxor    xmm0, xmm1          ; 다음 라운드 키
    movdqu  [%2], xmm0
%endmacro
```

AES

- aes128_invert_keys 함수

```
aes128_invert_keys:
    push    rbp
    mov     rbp, rsp

    ; dec[0] = enc[10]
    movdqu  xmm0, [rdi + 16*10]
    movdqu  [rsi], xmm0

    mov     ecx, 1
.inv_loop:
    cmp     ecx, 10
    jge     .last_key

    mov     eax, 10
    sub     eax, ecx            ; 10 - i
    imul    eax, 16
    movdqu  xmm0, [rdi + rax]
    aesimc  xmm0, xmm0          ; InvMixColumns

    mov     eax, ecx
    imul    eax, 16
    movdqu  [rsi + rax], xmm0

    inc     ecx
    jmp     .inv_loop
```

```
.last_key:
    ; dec[10] = enc[0]
    movdqu  xmm0, [rdi]
    movdqu  [rsi + 16*10], xmm0

    pop     rbp
    ret
```

AES

- aes128_encrypt_block 함수

```
aes128_encrypt_block:
    push    rbp
    mov     rbp, rsp

    movdqu  xmm0, [rdi]         ; state
    movdqu  xmm1, [rdx]         ; round key 0
    pxor    xmm0, xmm1          ; initial AddRoundKey

    mov     ecx, 1
.enc_loop:
    cmp     ecx, 10
    jge     .enc_last

    mov     eax, ecx
    imul    eax, 16
    movdqu  xmm1, [rdx + rax]
    aesenc  xmm0, xmm1
    inc     ecx
    jmp     .enc_loop
```

```
.enc_last:
    mov     eax, 10
    imul    eax, 16
    movdqu  xmm1, [rdx + rax]
    aesenclast xmm0, xmm1

    movdqu  [rsi], xmm0

    pop     rbp
    ret
```

AES

- aes128_decrypt_block 함수

```
aes128_decrypt_block:
    push    rbp
    mov     rbp, rsp

    movdqu  xmm0, [rdi]         ; state (ciphertext)
    movdqu  xmm1, [rdx]         ; dec round key 0 (enc[10])
    pxor    xmm0, xmm1          ; initial AddRoundKey

    mov     ecx, 1
.dec_loop:
    cmp     ecx, 10
    jge     .dec_last

    mov     eax, ecx
    imul    eax, 16
    movdqu  xmm1, [rdx + rax]
    aesdec  xmm0, xmm1
    inc     ecx
    jmp     .dec_loop
```

```
.dec_last:
    mov     eax, 10
    imul    eax, 16
    movdqu  xmm1, [rdx + rax]
    aesdeclast xmm0, xmm1

    movdqu  [rsi], xmm0

    pop     rbp
    ret
```

AES

결과 검증 – verify.sh

```
#!/usr/bin/env bash

if [ $# -ne 2 ]; then
  echo "Usage: $0 <plain-int> <key-int>"
  exit 1
fi

plain="$1"
key_int="$2"

if [ "$key_int" != "50462976" ]; then
  echo "[!] Warning: AES key is fixed in asm at 00..0F; key-int should be 50462976."
  echo
fi

echo "==================="
echo "  AES-NI (assembly)"
echo "==================="

printf "%s\n" "$plain" | ./aes-ni
echo

echo "==================="
echo "  OpenSSL reference"
echo "  (AES-128-ECB, key = 00..0F, no padding)"
echo "==================="
```

```
pt_hex=$(python3 - <<PY
import struct
n = int("$plain")
block = struct.pack("<I", n) + b"\x00"*12
print(block.hex())
PY
)

echo "Plain block (hex)  : $pt_hex"

ct_hex=$(python3 - <<PY
import struct, subprocess, binascii

n = int("$plain")
pt = struct.pack("<I", n) + b"\\x00"*12

key_hex = "000102030405060708090a0b0c0d0e0f"

p = subprocess.Popen(
    ["openssl", "enc", "-aes-128-ecb",
     "-K", key_hex,
     "-nopad", "-nosalt"],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
)
out, _ = p.communicate(pt)
print(binascii.hexlify(out).decode())
PY
)
```

```
dec_hex=$(python3 - <<PY
import subprocess, binascii

ct_hex = "$ct_hex"
ct = binascii.unhexlify(ct_hex)

key_hex = "000102030405060708090a0b0c0d0e0f"

p = subprocess.Popen(
    ["openssl", "enc", "-d", "-aes-128-ecb",
     "-K", key_hex,
     "-nopad", "-nosalt"],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
)
out, _ = p.communicate(ct)
print(binascii.hexlify(out).decode())
PY
)

echo "Decrypted (hex)    : $dec_hex"
echo
echo "비교 포인트:"
echo "  - AES-NI Encrypted block == OpenSSL Cipher block"
echo "  - AES-NI Decrypted block == OpenSSL Decrypted"
echo "  - Recovered int == original plain-int ($plain)"
```

AES

결과 검증 – verify.sh

```
oscar@oscar:~/asm/pj$ ./verify.sh 1234 50462976
====================
  AES-NI (assembly)
====================
AES-128 key (hex)    : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
Key first 4 bytes int: 50462976
Enter integer: Encrypted block (hex): AE 60 84 6D D3 79 58 32 F0 AF 89 63 87 B9 29 6E
Decrypted block (hex): D2 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Recovered int        : 1234

====================
  OpenSSL reference
  (AES-128-ECB, key = 00..0F, no padding)
====================
Plain block (hex)  : d20400000000000000000000000000
Cipher block (hex) : ae60846dd3795832f0af896387b9296e
Decrypted (hex)    : d20400000000000000000000000000

비교 포인트:
  - AES-NI Encrypted block == OpenSSL Cipher block
  - AES-NI Decrypted block == OpenSSL Decrypted
  - Recovered int == original plain-int (1234)
```

결론

- 특수 목적 명령어는 시스템 성능·보안 핵심 요소
- 소프트웨어보다 빠르고 안전한 하드웨어 처리
- 커널·VM·AES 등 분야별 사용 사례 이해
- Assembly 분석이 시스템 구조 이해로 이어짐
- 다른 생소한 명령어를 보더라도 두렵지 아니함

THANKS FOR PLAYING

END