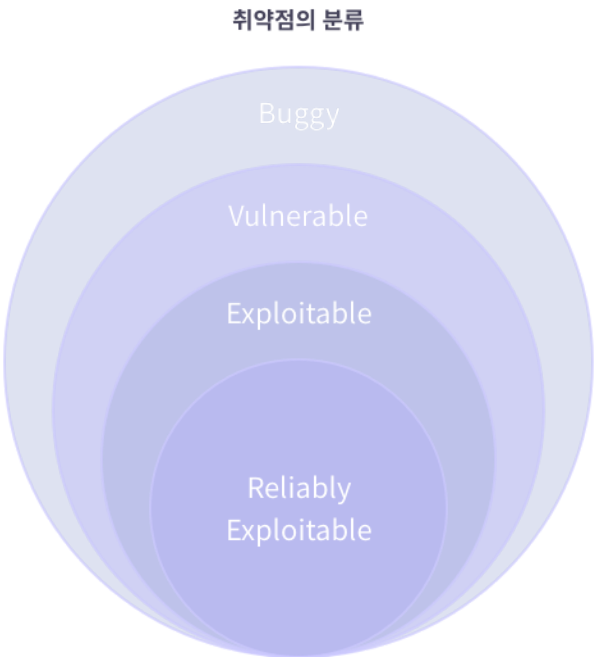
DreamHack - System Exploitation Fundamental

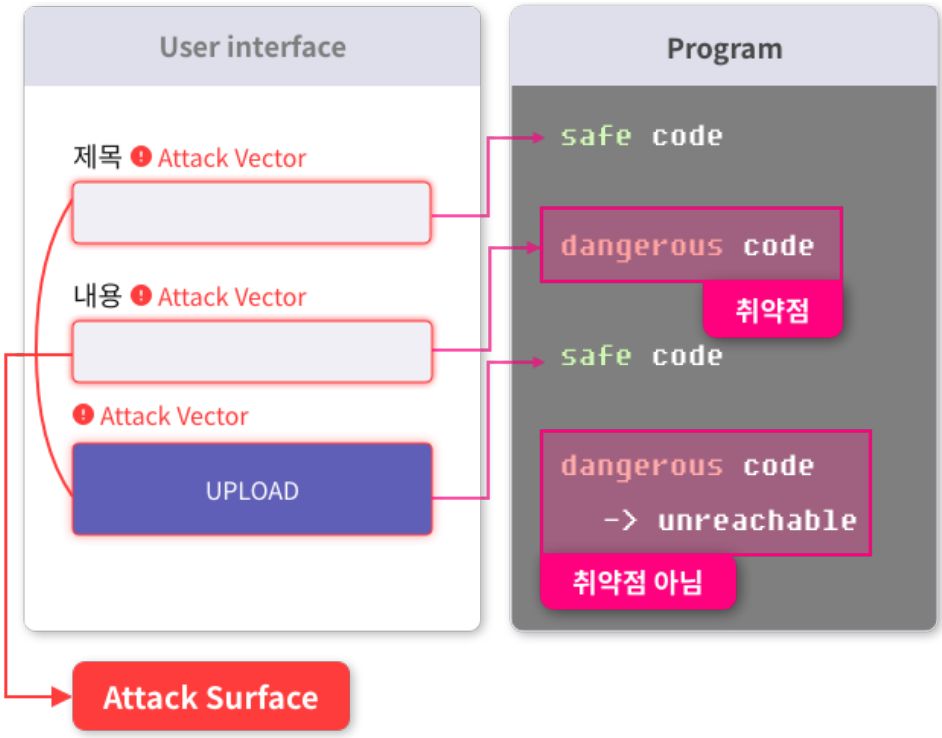
취약점의 분류



- 익스플로잇 : 취약점을 이용해 공격자가 의도한 동작을 수행하게 하는 코드 혹은 이를 이용한 공격 행위
- 소프트웨어 버그 (Bug) : 프로그래머가 의도하지 않은 동작을 수행
- 소프트웨어 취약점(Vulnerability) : 보안에 영향을 미칠 수 있는 버그
- 익스플로잇 가능한 취약점(Exploitable Vulnerability) : 공격자가 의도한 동작을 수행할 수 있는 버그
- 안정적으로 익스플로잇 가능한 취약점(Reliably Exploitable Vulnerability) : 익스플로잇이 가능한 취약점 중 매우 높은 확률로 공격에 성공할 수 있는 버그

Attack Vector

- 공격자가 소프트웨어와 상호 작용할 수 있는 곳
- Attack Surface : Attack Vector들의 집합



취약점의 종류

메모리 커럽션 취약점 : C/C++과 같은 저수준 언어에서 메모리를 조작해 공격

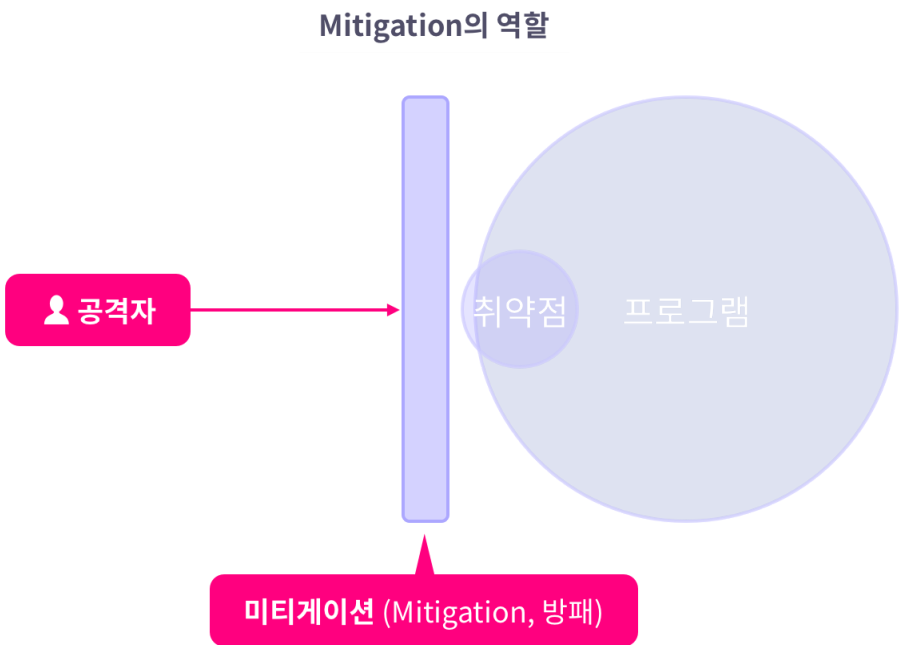
- Buffer Overflow (BOF) : 프로그래머가 할당한 크기의 버퍼보다 더 큰 데이터를 입력받아 메모리의 다른 영역을 오염시킬 수 있는 취약점
- Out-Of-Boundary (OOB) : 버퍼의 길이 범위를 벗어나는 곳의 데이터에 접근할 수 있는 취약점
- Off-by-one : 경계 검사에서 하나 더 많은 값을 쓸 수 있을 때 발생하는 취약점
- Format String Bug : printf나 sprintf와 같은 함수에서 포맷 스트링 문자열을 올바르게 사용하지 못해 발생하는 취약점
- Double Free / Use-After-Free : 동적 할당된 메모리를 정확히 관리하지 못했을 때 발생하는 취약점
- etc..

로지컬 버그 : 메모리를 조작할 필요 없이 공격

- Command Injection : 사용자의 입력을 셸에 전달해 실행할 때 정확한 검사를 실행하지 않아 발생하는 취약점
- Race Condition : 여러 스레드나 프로세스의 자원 관리를 정확히 수행하지 못해 데이터가 오염되는 취약점. 발생 원인과 공격 방법에 따라 메모리 커럽션 취약점으로도, 로지컬 취약점으로도 분류할 수 있음
- Path Traversal : 프로그래머가 가정한 디렉토리를 벗어나 외부에 존재하는 파일에 접근할 수 있는 취약점
- etc..

Mitigation

- 취약점의 존재 여부와는 무관하게 프로그램을 보호하는 방법



버퍼 오버플로우

버퍼 오버플로우가 무엇인지는

해커 지망자들이 알아야 할 Buffer Overflow Attack 의 기초 - 정리

에 자세히 정리되어 있으니, 실제 코드를 보며 취약점들을 알아가도록 하자.

- 스택 버퍼 오버플로우

```
// stack-1.c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    char buf[16];
    gets(buf);

    printf("%s", buf);
}
```

gets 함수에서 취약점 발생.

buf에 의해 스택에는 16byte buffer가 할당되지만, gets 함수는 개행 문자가 나오기 전까지 입력된 내용을 인자로 전달된 buffer에 저장하여 buffer의 크기 이상의 정보가 메모리를 오염시킬 수 있다. 이것으로 return address를 바꾸어 쉘코드를 실행시키는 등 이용할 수 있다.

```
// stack-2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_auth(char *password) {
    int auth = 0;
    char temp[16];

    strncpy(temp, password, strlen(password));

    if(!strcmp(temp, "SECRET_PASSWORD"))
        auth = 1;

    return auth;
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ./stack-1 ADMIN_PASSWORD\n");
        exit(-1);
    }

    if (check_auth(argv[1]))
        printf("Hello Admin!\n");
    else
        printf("Access Denied!\n");
}
```

프로그래머의 실수에 의해 취약점이 발생하는 경우.

strncpy 함수를 사용하였지만, temp의 길이로 복사하는 것이 아닌 password의 길이만큼 데이터를 옮기므로 메모리를 오염시킬 수 있다. 따라서

```
if(!strcmp(temp, "SECRET_PASSWORD"))
        auth = 1;
```

의 실행과는 상관없이, 메모리를 조작하여 auth의 값을 바꿀 수 있으므로 취약한 프로그램이 된다.

```
// stack-3.c
#include <stdio.h>
#include <unistd.h>
int main(void) {
    char win[4];
    int size;
    char buf[24];

    scanf("%d", &size);
    read(0, buf, size);
    if (strncmp(win, "ABCD", 4)){
        printf("Theori{-----------redeacted---------}");
    }
}
```

사용자가 올바른 값만 넣을 것이라 생각하여 취약점이 발생하는 경우이다.

buf의 크기는 24byte로 고정되어 있으나, size로 입력받을 수 있는 값에는 제한이 없으니 BOF가 가능하게 된다.

```
// stack-4.c
#include <stdio.h>
int main(void) {
  char buf[32] = {0, };
  read(0, buf, 31);
  sprintf(buf, "Your Input is: %s\n", buf);
  puts(buf);
}
```

이 또한 프로그래머의 실수에 의해 취약점이 발생한다.

read 함수에 buf의 크기만큼 입력받을 크기를 정해주었지만, sprintf에서 먼저 추가해 준 문자열의 크기를 생각하지 않았다. 이 문자열의 크기만큼 buf의 버퍼를 넘어 입력이 가능하므로, 그 부분에서 BOF가 가능하다.

- 힙 오버플로우

```
// heap-1.c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    char *input = malloc(40);
    char *hello = malloc(40);

    memset(input, 0, 40);
    memset(hello, 0, 40);

    strcpy(hello, "HI!");
    read(0, input, 100);

    printf("Input: %s\n", input);
    printf("hello: %s\n", hello);
}
```

스택 영역에서가 아닌, 힙 영역에서 오버플로우가 일어난다.

read 함수에 의해 input 버퍼의 크기인 40byte보다 큰 100byte까지의 데이터를 읽어올 수 있으므로 input 버퍼를 넘어 heap 영역의 메모리를 오염시킬 수 있다.

System Exploitation Fundamental

Hackers' Playground, DreamHack

https://dreamhack.io/lecture/curriculums/2