History of Buffer Overflow

Last Update: 2008년 7월 3일

Written by Jerald Lee

Contact Me: lucid78@gmail.com

작성자의 말:

본 문서는 지난 20년간 발전되어온 버퍼 오버플로우 공격 및 방어 기술을 시대 순에 따라 살펴본 것입니다. 원래 논문용으로 만들다가 학문적인 가치가 전혀 없다는 교수님의 말씀에 그냥 접어버렸던 문서인데 버리기가 아까워 이렇게 살짝 다듬어서 내놓게 되었습니다. 이 문서를 읽으시기 전 아래의 주의 사항을 숙지하시기 바랍니다.

1. 본 문서는 오직 버퍼 오버플로우 기술에 대해서만 다룹니다. 그러므로 본 문서는 각 시대별 보안 상황을 나타내는 지표로써는 적합하지 않습니다.
2. 각 시대별로 잘 알려지고 유명한 기술들만을 뽑았습니다. 즉 어디까지나 저의 주관적인 해석에 따라 발췌되었으므로 빠진 기술이 있을 수 있으며 몇몇 기술의 경우 고의적으로 배제된 것도 있습니다.
3. 포맷 스트링 기술의 경우 버퍼 오버플로우에 속하느냐 속하지 않느냐에 대한 논란의 여지가 있을 수 있습니다. 최근의 많은 버퍼 오버플로우 공격들이 포맷 스트링 기술과 결합한 형태를 보이기 때문에 본 문서에서는 포맷 스트링 기술을 버퍼 오버플로우 기술 중의 하나로 다루었습니다.
4. 참고 자료 외에도 많은 문서를 참조하였으나 일일이 기술하지 못하였습니다. 원 저작자들의 많은 양해 부탁 드립니다. 또한 Hacking Document Project[1]에 올라와 있는 문서 역시 참조하였습니다.

Special Thanks to: Xpl017Elz

복잡한 코드의 설명과 같은 난이도 있는 부분의 경우 **Xpl017Elz**의 많은 도움을 받았습니다.

문서의 내용 중 틀린 곳이나 수정할 곳이 있으면 연락해 주시기 바랍니다.

[1] http://hdp.null2root.org

Abstract

2008년은 버퍼 오버플로우 취약점을 이용한 Morris Worm이 발생한 지 20주년이 되는 해이다. Aleph1의 유명한 스택 오버플로우 기사 발표 이후 봇물처럼 쏟아지기 시작한 버퍼 오버플로우 관련 공격 및 방어 기술들은 20년에 걸쳐서 발전하게 되며 각 시대별로 특징을 가지고 있다. 본문에서는 지난 20여 년간 발전되어 온 버퍼 오버플로우 취약점 기반 공격 및 방어 기술에 대해 살펴보고 이들 기술이 발전되는 과정에서 보이는 특징에 대해 조사하였다.

차 례

그림 차례

1. 서론

2008년은 버퍼 오버플로우 취약점을 이용한 Morris Worm이 발생한 지 20주년이 되는 해이다. 최초의 버퍼 오버플로우 가능성은 1973년경 C언어의 데이터 무결성 문제[2]로 그 개념이 처음 알려졌으나[1] 1988년 인터넷을 통해 확산된 최초의 Worm인 Morris Worm에 의해 버퍼 오버플로우 문제의 심각성이 널리 알려지게 되었다. 1995년에 들어 버퍼 오버플로우 취약점을 이용한 공격 코드들이 하나 둘씩 발표되기 시작하면서 본격적인 취약점 연구가 이루어지기 시작했고 1996년 Aleph1의 "Smashing the stack(for Fun and Profit)" 이라는 기사가 Phrack 49호에 발표되면서 '버퍼 오버플로우의 시대' 라고 불릴 정도로 많은 공격 코드들이 발표되게 된다. 이 때부터 시작된 버퍼 오버플로우에 대한 공격/방어 진영의 싸움은 지금까지도 계속 이어지고 있다. 본 문서에서는 지난 20여 년간 발전되어 온 버퍼 오버플로우 취약점 기반 공격 및 방어 기술들을 살펴보고 이들 기술이 발전되는 과정에서 보이는 특징에 대해 조사하였다.

[2] 이 부분에 관한 논문 및 문서는 찾을 수 없었음. 관련 자료가 있다면 연락 부탁합니다.

2. 버퍼 오버플로우 공격의 원리

버퍼 오버플로우란 버퍼를 넘치게 하여 프로그램의 실행 흐름을 변경함으로써 임의의 코드를 실행하게 하는 기술이다. 버퍼의 경계 값을 검사하지 않는 함수를 사용할 때 이 함수가 사용자의 입력 값을 다룬다면 공격자는 버퍼를 넘치는 값을 주입하여 프로그램의 실행 흐름을 변경할 수 있다.

버퍼의 경계 값을 검사하지 않는 대표적인 함수들은 아래와 같다[2].

strcpy()
strcat()
gets()
fscanf()
scanf()
sprintf()
sscanf()
vfscanf()
vsprintf
vscanf()
vsscanf()
streadd()
strecpy()
strtrns()

그림 1. 버퍼 오버플로우에 취약한 함수들

3. 버퍼 오버플로우 공격 및 방어 기술의 발전

이번 절에서는 20여 년간 발전되어온 주요 공격 및 방어 기술을 시대 순으로 살펴보고 각 시대 별 특징을 살펴보았다.

3.1. 1996년 ~ 1999년

이 기간은 버퍼 오버플로우 공격의 시대라 불릴 정도로 버퍼 오버플로우 취약점에 대한 많은 연구가 이루어졌다. 많은 연구자들에 의해 새로운 공격 기술과 방어 기술이 발표되었으며 이 때 연구된 기술들을 기반으로 하는 많은 변형된 기술들이 이후에 등장하게 된다. 이들 기술들은 서로 역학적인 관계를 형성하며 발전하는 형태를 보인다.

3.1.1. Stack Overflow

1996년 11월, Phrack[3] 49호에 Aleph1은 버퍼 오버플로우 취약점을 가지는 프로그램을 공격하는 자세한 방법을 설명한 기사를 발표하는데[3], 현재 이 기사는 많은 보안 전문가들이 버퍼 오버플로우의 고전으로 여기고 있다. 이 기사는 버퍼의 경계를 검사하지 않는 strcpy()와 같은 함수에 의해 문자열이 복사될 때 버퍼를 넘치게 하여 임의의 코드를 실행할 수 있음을 보여주는데 스택의 특성을 이용하므로 스택 오버플로우라고 부르는 것이 일반적이다. 버퍼 오버플로우를 공부할 때 가장 처음 접하게 되는 기술이기도 하다.

아래의 그림 2는 스택 오버플로우 취약점을 가진 코드를 나타낸다.

```
int main(int argc, char *argv[])
{
    char buf[10];
    strcpy(buf, argv[1]);
    return 0;
}
```

그림 2. 스택 오버플로우 취약점을 가진 예제 코드

공격자는 'argv[1]' 변수를 통해 공격 코드를 주입하고 그림 3처럼 덮어 쓰여진 RET에 의해 공격 코드가 실행된다.

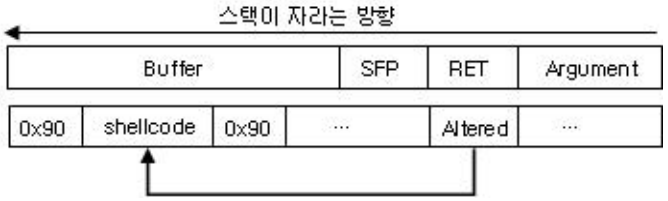

그림 3. 스택 오버플로우 공격

[3] http://www.phrack.org

3.1.2. Non-executable Stack

1997년에 Solar Designer는 스택에서 어떤 코드도 실행되지 못하도록 하는 기능을 가진 Non-executable Stack을 리눅스 커널 패치의 형태로 개발하여 배포하는데, 이를 적용할 경우 [3]에서 제시한 스택 안에 공격 코드를 주입하여 실행시키는 공격은 실패하게 된다. 이 기술은 Solar Designer에 의해 이후 OpenWall Project[4]로 발전하게 된다.

3.1.3. Return Into Libc

1997년 8월, Solar Designer는 Bugtrack[5]에서 논의되던 'Return Into Libc'(이하 RTL) 기술을 이용한 공격 코드를 공개하면서 자신의 Non-executable Stack 패치를 수정한다. RTL은 함수의 Return Address(이하 RET)를 덮어써서 공유 라이브러리에 존재하는 system()과 같은 함수를 가리키게 하고 함수 실행에 필요한 매개 변수를 전달하여 실행하는 공격법이다. 메모리에 적재된 공유 라이브러리는 스택에 존재하는 것이 아니므로 Non-executable Stack을 우회하는 것이 가능하다.

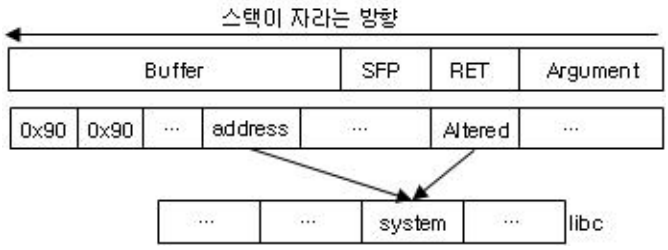

그림 4. Return Into Libc 공격

Solar Designer는 그림 4처럼 RET를 메모리에 적재된 system() 함수를 가리키게 하고, 환경변수에 저장한 "/bin/sh" 의 주소를 ebp + X(레드햇 리눅스 8.0 기준으로 ebp+8에서 인자 값을 읽는다)에 위치시켜 system() 함수에 전달함으로써 shell을 획득할 수 있음을 증명하였다. 만약 공격자가 setuid() 함수를 이용할 경우 root 권한의 shell도 획득 가능함을 증명하였다.

이 취약점을 막기 위해 Solar Designer는 공유 라이브러리들의 기본 주소를 mmap() 함수를 사용하여 '0'을 포함하는 주소로 변경하였다. 예를 들어, system()의 주소는 0x00401037인 식이다. 일반적으로 공격 코드는 아스키 문자열의 형태로 전달되므로 '0'을 문장의 끝으로 인식하기 때문에 그림 4의 공격 방법으로는 RET를 덮어쓸 수 없게 된다.

3.1.4. Defeating Non-executable Stack 패치

[3.1.3]에서 살펴본 Solar Designer의 패치로 인해 공유 라이브러리들의 주소들은 16MB

[4] http://www.openwall.com/
[5] http://www.securityfocus.com/archive/1

미만, 즉 '0'이 포함되는 주소 값을 가지게 되었다. 1998년 1월에 Nergal은 이것을 우회하는 두 가지 방법을 제안한다[6].

가. PLT(Procedure Linkage Table) 이용

리눅스에서 공유 라이브러리는 PLT와 GOT(Global Offset Table)을 이용하는데, PLT는 프로그램이 호출하는 모든 함수를 나열하고 있는 테이블이고 GOT는 프로그램 실행 후 libc.so 내의 실제 함수 주소가 저장되는 곳이다. 가장 처음 공유 라이브러리 내의 함수가 호출되었을 때 동적 링커는 먼저 PLT를 살펴본다. PLT는 실제 호출될 함수를 나타내는 값을 _dl_runtime_resolve 함수의 인자로 넘기고 _dl_runtime_resolve 함수는 전달된 인자 값을 사용하여 호출된 함수의 실제 주소를 구한 후 GOT에 저장한 뒤 호출된 함수로 점프한다. 이후 동일한 함수가 다시 호출되면 동적 링커는 GOT에 저장되어 있는 호출된 함수의 실제 주소로 바로 점프하게 된다.

공격자는 대상 프로그램의 취약한 함수를 이용하여 RET를 strcpy() 함수의 PLT 주소로 변경한다. PLT 주소에는 Null String이 포함되어 있지 않으므로 strcpy() 함수가 실행되는데, 이 함수는 환경변수에 저장된 shellcode를 데이터 세그먼트 영역에 복사한다. strcpy() 함수가 완료된 후 RET이 복사된 데이터 세그먼트 영역의 shellcode를 가리키면서(DEST1이 pop된다) shellcode가 실행되게 된다.

아래 그림 5는 PLT를 이용한 RTL 공격을 보여준다.

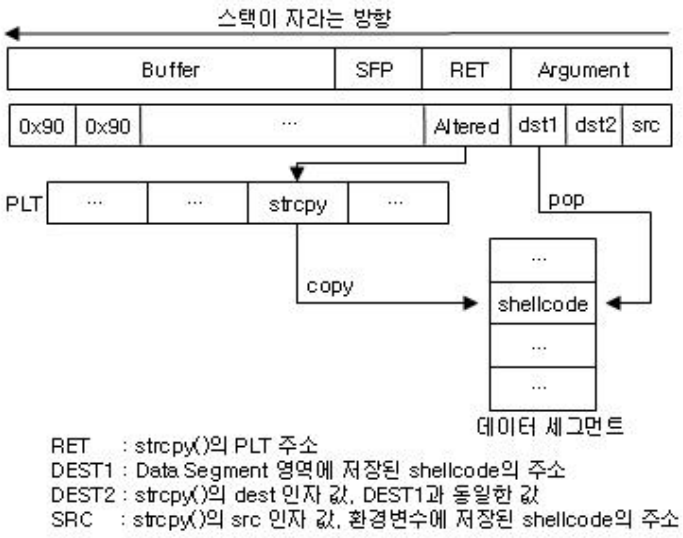

그림 5. Defeat Non-executable Stack Patch with PLT

나. strcpy()를 두 번 호출하여 GOT를 덮어쓰기

상기하였듯이 두 번째부터의 동일한 함수 호출 시에는 GOT에 저장된 주소로 점프하게 되므로 첫 번째 strcpy() 실행 시 GOT에 system()의 주소가 쓰여지게 한다. 그리고 두 번째로 strcpy()가 호출될 때는 GOT에 저장된 system()의 주소로 점프하게 되고

환경변수에 저장된 인자 값을 실행하게 된다.
아래 그림 6은 GOT를 덮어쓰는 RTL 공격을 보여준다.

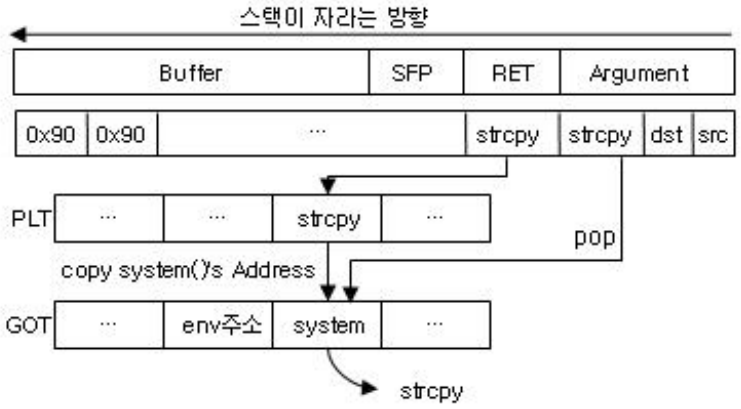

그림 6. Defeat Non-executable Stack Patch with double strcpy()

3.1.5. Stack Guard

Non-executable Stack을 우회하는 기술이 발표된 때와 비슷한 시기에 Stack Guard라는 버퍼 오버플로우 방어 기술이 발표된다[7]. Stack Guard는 프로그램 실행 시 버퍼 오버플로우 공격을 탐지하도록 gcc 컴파일러(version 2.7.2.2)의 확장으로 만들어졌다. 즉, 컴파일러가 프로그램의 함수 프롤로그 시에 RET 앞에 Canary 값을 주입하고 에필로그 시에 Canary 값이 변조되었는지의 여부를 확인하여 버퍼 오버플로우 공격을 탐지하게 된다.
아래 그림 7은 Stack Guard를 이용하여 컴파일 된 스택의 모습을 보여준다.

그림 7. Stack Guard

그림 7에서 볼 수 있듯이 그림 3처럼 버퍼부터 RET까지 덮어쓰는 전통적인 형태의 공격법은 Canary 값을 덮어쓸 수 밖에 없으므로 버퍼 오버플로우가 발생했음을 탐지할 수 있다. 단조로운 값의 Canary를 사용할 경우 추측에 의한 우회가 가능할 수 있으므로 Canary는 프로그램 시작 시간에 crt0 라이브러리를 이용하여 무작위로 생성된다.
Stack Guard는 이후 몇 차례의 변경이 있었는데 Null Canary, Terminator Canary, XOR Random Canary가 그것이다. Null Canary, Terminator Canary는 Canary로 사용될 값에 Null 문자나 Terminator 문자가 들어가게 하여 strcpy() 같은 문자열 복사 함수가 실행될 때 해당 문자에 의해 실행이 종료되어 버리도록 한다. 이후에 살펴보겠지만 Bulba와 Kil3r이라는 두 명의 해커가 Stack Guard를 우회할 수 있는 아이디어를 Crispin Cowan에게 이야기하였고 이에 XOR Random Canary를 포함하는 Stack Guard 1.21가

발표되게 된다[8]. Stack Guard 1.21은 함수가 호출될 때 RET와 Random Canary를 XOR한 값을 메모리에 저장한 후 RET 앞에 위치시킨다(Canary). 이후 함수가 return될 때 메모리에 저장되어있던 Canary와 스택 상의 RET 값을 XOR 연산한 후 이 값을 스택 상의 XOR된 Canary 값과 비교하여 일치하면 return이 완료된다. 만약 RET를 공격자가 변조시켰다면 XOR된 Canary 값과 비교하여 탐지할 수 있다.

3.1.6. Windows 버퍼 오버플로우

이 당시 모든 버퍼 오버플로우 공격이 Linux에 집중되어 있던 것과는 달리 1998년 4월에 CDC(Cult of Dead Cow) 멤버인 Dildog가 Windows 운영체제에서의 버퍼 오버플로우 방법을 기술한 기사를 발표한다[23]. 이 기사에 의해 소스가 공개되어 있지 않아 리눅스와는 달리 비교적 안전하다고 생각되었던 Windows 역시 버퍼 오버플로우에 취약점에 대해 결코 안전하지 않다는 것이 알려지게 된다. Windows에서의 버퍼 오버플로우 공격은 리눅스에서의 버퍼 오버플로우 공격과 기본 원리는 동일하나 shell을 실행시키기 위해 필요한 API를 가지고 있는 몇몇 dll 파일을 적재하는 코드가 포함되어야 한다는 점에서 다소 차이점이 있다.

3.1.7. Random Stack

1998년 8월에 버퍼 오버플로우 공격에 대한 방어 기술의 하나로 Random Stack이 제안된다[30]. 이 기술은 공격자가 RET를 변조하기 위해서는 오버플로우 되는 버퍼의 주소를 알아내는 것이 필요하다는데 착안을 두어, 프로그램이 실행될 때마다 스택이 서로 다른 주소에 위치하도록 하여 버퍼의 주소를 알아내기 힘들게 한다. 이 기사는 스택을 비고정적으로 위치시키기 위해 /dev/urandom과 alloca()를 쓸 것을 제안하였으며 또한 RTL과 같은 공격으로부터의 방어를 위해 공유 라이브러리의 주소 역시 무작위로 할당되도록 할 것을 제안하였다.

3.1.8. Heap Overflow

1999년 1월에는 스택에 집중되어 있던 버퍼 오버플로우 공격이 Heap 메모리에서도 가능하다는 것을 증명하는 기사가 발표되었는데[9] 이 기사에서는 솔라리스의 protect_stack이나 리눅스의 Stack Guard와 같은 스택 보호 기술이 Heap 메모리에도 필요함을 언급하였다. 그리고 malloc()에 의해 할당되는 Heap 영역 외에도 static 지시자에 의해 할당되는 bss 영역에서도 동일하게 버퍼 오버플로우가 일어날 수 있음을 증명하였다. 또한 최초의 jmpbuf 덮어쓰기 기술이 소개되는데, setjmp() 함수를 이용해 현재의 명령어, 스택 포인터, 다른 레지스터를 jmp_buf에 저장한 후 longjmp() 함수로 setjmp()에 의해 저장된 jmpbuf를 복구한 다음 jmpbuf를 덮어씀으로써 longjmp() 함수가 호출될 때 shellcode를 실행할 수 있다.

```
#define BUFSIZE 16
```

```
#define OVERSIZE 8

int main()
{
    u_long diff;
    char *buf1 = (char *)malloc(BUFSIZE);
    char *buf2 = (char *)malloc(BUFSIZE);

    diff = (u_long)buf2 – (u_long)buf1;
    printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2, diff);

    memset(buf2, 'A', BUFSIZE-1);
    buf2[BUFSIZE-1] = '\0';

    printf("before overflow : buf2 = %s\n", buf2);
    memset(buff1, 'B', (u_int)(diff + OVERSIZE));
    printf("after overflow : buf2 = %s\n", buf2);

    return 0;
}
```

그림 8. Heap 오버플로우 취약점을 가진 예제 코드

그림 8은 Heap 오버플로우 취약점을 가진 예제 코드이며 이 프로그램의 실행 결과는 아래와 같다.

```
buf1 = 0x804e00, buf2 = 0x804eff0, diff = 0xff0 bytes
before overflow : buf2 = AAAAAAAAAAAAAAA
after overflow : buf2 = BBBBBBBBAAAAAAA
```

그림 9. 프로그램 실행 결과

그림 9에서 memset() 함수가 실행되면서 buf1을 넘어 인접한 buf2의 영역을 겹쳐 쓴 것을 확인할 수 있다. 동일한 방법으로 bss 영역에서도 오버플로우가 가능함을 증명하였다.

3.1.9 strlcpy() and strlcat()

1999년 6월, OpenBSD 진영에서 strncpy()와 strncat()을 대체할 새로운 함수인 strlcpy(), strlcat()을 개발한다[10]. 이는 버퍼 오버플로우 취약점을 피하기 위해 개발자들이 strcpy(), strcat() 대신 strncpy(), strncat()에 의존하였기 때문이었다.

strncpy(), strncat()의 경우 자동으로 Destination의 문자열을 Null String으로 마치게 해 주지만 사용법의 모호함으로 인한 유지보수의 어려움과 Source 문자열의 길이가 Destination 문자열의 길이보다 클 때 여전히 오버플로우가 일어날 수 있다는 문제점을 가지고 있었다. strlcpy()와 strlcat()은 명확한 사용법을 가지고 있고 버퍼 오버플로우를 방어할 수 있으며 무료로 이용할 수 있다는 장점에도 불구하고 아직까지 BSD 계열의 유닉스 외의 운영체제에는 기본으로 포함되지 않고 있다.

3.1.10. Stack Shield

Stack Shield 역시 GNU C 컴파일러의 확장으로서 개발되었으며[11] RET를 보호하는 것을 주 목적으로 한다. Stack Shield는 함수의 프롤로그 때 RET를 'Global RET Stack'이라는 특수 스택에 저장하고 함수의 에필로그 시에 'Global RET Stack'에 저장된 RET 값과 스택의 RET 값을 비교하여 일치하지 않으면 프로그램을 종료시킨다.

Stack Shield 역시 몇 차례의 변경이 있었는데 0.6 버전부터 함수 포인터를 보호하는 기능이 추가된다. Stack Shield는 함수 포인터가 오직 .text 영역만을 가리키도록 한다. 일반적으로 공격 코드는 .data 영역에 주입되므로 함수 포인터를 이용한 공격을 막을 수 있다.

3.1.11. Project Omega

1999년 8월, corezine[6] volume2에서 Lamagra 라는 해커가 shellcode가 필요 없는 버퍼 오버플로우 공격법을 개발하기 위한 Omega Project를 시작하는데[28] 기술적으로 봤을 때 Solar Designer의 RTL 공격 기술과 동일하다. 재미있는 것은 Lamagra는 RTL이라는 공격 기술이 이미 존재한다는 걸 모르는 상태에서 독립적으로 연구한 것 같아 보이는데, 결과물은 RTL과 동일하다는 것이다. 10월에 발표된 corezine volume3에서[29] 공격 코드를 완성하고 이 프로젝트를 종료한다.

3.1.12. Frame Pointer Overwrite

1999년 9월, Phrack 55호에 klog의 "The Frame Pointer Overwrite"라는 기사가 발표된다[12]. 이 기사의 핵심은 Saved Frame Point(이하 SFP)의 단지 1바이트 조작만으로도 프로그램의 실행 흐름을 바꾸는 것이 가능하다는 것이다. 이 기술은 SFP의 마지막 1바이트를 덮어써서 eip가 버퍼에 주입된 shellcode를 가리키는 주소가 되도록 조작한다.

리눅스에서는 호출된 함수의 에필로그 시에 leave, ret의 두 명령이 실행되는데 leave 명령어는 아래와 같이 두 개의 명령어로 구성되어 있다.

```
mov     ebp, esp
pop     ebp
```

[6] http://www.corezine.de/

그림 10. leave 명령어

그림 10처럼 leave 명령어에 의해 ret이 실행되기 전 1바이트가 조작된 ebp(sfp)값이 esp에 들어가게 된다. 이후 ebp가 pop되면 esp가 4바이트 증가하게 되어 결국 esp+4의 값이 eip에 들어가게 된다. 즉 eip는 esp+4 에 저장된 값을 RET으로 인식하여 버퍼에 주입된 shellcode를 실행하게 된다. 공격자의 입장에서는 esp+4를 예측하여 shellcode가 저장된 주소 값을 위치시키면 되는 것이다.

```
#include <stdio.h>

void func(char *sm) {
    char buffer[256];
    int i;

    for(i=0; i<=256; i++)
        buffer[i] = sm[i];
}

void main(int argc, char *argv[])
{
    if(argc < 2) {
        printf(“missing argv\n”);
    exit(-1);
}
func(argv[1]);
}
```

그림 11. 1바이트 오버플로우 취약점을 가지는 코드 예제

그림 11은 전형적인 1바이트 오버플로우 취약점을 가지는 예제 코드이며 오버플로우가 일어난 후의 스택의 모습은 아래와 같다.

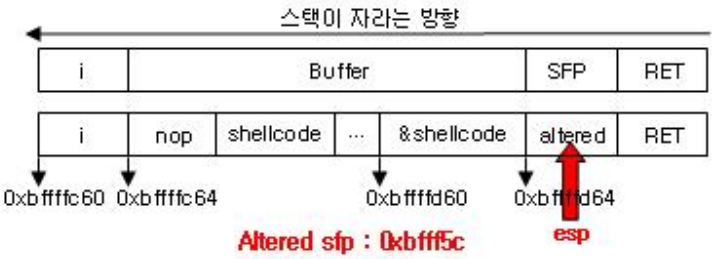

그림 12. Frame Point Overflow

오버플로우가 발생한 뒤 esp는 조작된 sfp를 가리키게 되고 이 값은 0xbffffd5c를 가진다(5c가 오버플로우에 의해 덮어씌워진 부분이다). 이제 컴파일러는 leave, ret을 실행하게 되는데 mov ebp, esp 명령에 의해 esp는 0xbffffd5c로 변경되고 pop ebp 명령에 의해 esp는 4바이트 증가하여 0xbffffd60이 된다. 이제 ret이 실행되면서 eip는 0xbffffd60, 즉 &shellcode를 실행하게 된다. 상기한 Stack Guard의 경우 RET 앞에 Canary를 주입하기 때문에 SFP가 변조되었는지의 여부는 알 수 없으므로 Stack Guard를 우회하는 것이 가능하다(이 문서 발표 전에 Stack Guard는 Canary를 SFP 앞에 주입하도록 변경되었다). 또 Stack Shield 역시 우회할 수 있지만 취약한 버퍼가 함수 내에서 가장 먼저 선언되어야 한다는 조건이 성립하여야만 한다.

3.2. 2000년

2000년 역시 버퍼 오버플로우 취약점에 대한 연구가 여전히 주류를 이루게 된다. 하지만 2000년 후반기에 발표되는 포맷 스트링 취약점이 버퍼 오버플로우 취약점을 대체하는 하나의 흐름으로 자리잡게 된다. 또 이 시기에 발표된 Heap 오버플로우 공격 기술을 시작으로 Heap에 대한 버퍼 오버플로우 연구가 증가하여 이후 Windows의 Internet Explorer 관련 취약점이 많이 연구되게 된다. 이 시기를 대표하는 가장 큰 특징으로 PaX를 들 수 있는데 PaX에 구현된 대부분의 방어 기술은 최근의 운영체제들에 구현된 방어 기술의 모태 역할을 하게 된다.

3.2.1. Static Source Code Analysis

2000년 2월에는 버퍼 오버플로우 취약점에 대응하기 위한 새로운 방식의 접근이 시도된다[13]. 이는 정적 소스코드 검사를 기반으로 하는 버퍼 오버플로우 공격 자동탐지 기술이다. 이 기술은 C 언어의 문자열을 추상 데이터 타입으로, 버퍼를 숫자 쌍으로 구분하여 소스코드를 분석한다. 하지만 false negative와 false positive가 많이 발생하고, 분석하는 소스코드가 클수록 오탐 확률이 더욱 높아진다. 탐지율이 낮음에도 불구하고 이 기술에서 사용된 정적 소스코드 검사는 향후 버퍼 오버플로우 방지를 위한 주요 방법 중의 하나로 자리매김하게 되어 ITS4[35], Cqual[34], RATS[36], Cyclone[38], CCured[39]와 같은 분석 도구가 개발되게 된다.

3.2.2. Bypass Stack Guard and Stack Shield

2000년 5월, Phrack 56호에 Bulba와 Kil3r라는 두 명의 해커가 Stack Guard 와 Stack Shield를 우회하는 여러 가지 기술을 발표하였다[14].

가. 포인터 변수 덮어쓰기

이 기술은 Frame Pointer Overwrite와 비슷하며 매우 프로그램 의존적이다. 버퍼가 선언되기 전 포인터 변수가 하나 선언되어 있을 때 버퍼를 넘치게 하여 포인터 변수를 덮어써서 이것이 RET를 가리키게 한 다음 strcpy() 함수와 같은 취약한 문자열 복사 함수를 통해 RET를 덮어쓰는 것이다.

```
int f(char **argv) {
    int pipa;
    char *p;
    char a[30];

    p=a;

    printf("p=%x\t -- before 1st strcpy\n", p);
    strcpy(p, argv[1]);
    printf("p=%x\t -- after 1st strcpy\n", p);
    strncpy(p, argv[2], 16);
    printf("After second strcpy\n");
}

main(int argc, char **argv)
{
    f(argv);
    execl("anyprogram", "" ,0);
    printf("End of program\n");
}
```

그림 13. 버퍼 오버플로우 취약점을 가지는 예제 코드

그림 13은 버퍼 오버플로우 취약점을 가지는 예제 코드이다. 오버플로우가 발생된 후 스택의 모습은 아래와 같다.

그림 14. Bypass Stack Guard and Stack Shield with pointer

그림 14에서 볼 수 있듯이 공격자는 strcpy() 함수 실행 시 p 포인터의 값을 RET 주소 값으로 덮어쓴다. 이후 strncpy() 함수가 실행되면서 p 포인터에 써지는 값은 RET를 덮어쓰게 되고 이 값은 &shellcode가 된다. 프로그램 실행이 끝나면 shellcode가 실행된다.

나. Stack Shield 우회하기

Stack Shield의 경우 복사된 RET list를 이용해 RET의 무결성을 검사하기 때문에 위의 방법으로는 우회할 수가 없으므로 atexit() 함수 또는 exit() 함수를 공격하는 기술을 제시하였다. 이 기술은 p 포인터의 값을 atexit()함수와 exit() 함수에서 동일하게 사용하는 fnlist 구조체 내의 _fini 함수 또는 _dl_fini 함수의 주소로 바꾸는 것이다. 대부분의 프로그램이 실행을 완료한 후 exit()를 호출하므로 이 때 조작된 _fini 함수 또는 _dl_fini 함수가 실행되고, shellcode가 실행되게 된다. 이 시기의 Stack Guard 역시 exit() 함수를 사용하고 있었으므로 klog는 해당 취약점을 Stack Guard 개발자 중 한명인 Perry Wagle에게 알렸다. 이후 Stack Guard는 _exit() 함수를 호출하도록 수정되었다.

다. Stack Guard + Non-executable Stack 패치 우회하기

만약 Stack Guard에 Solar Designer의 Non-executable Stack 패치가 적용되어 있다면 위의 기술은 통하지 않는다. 이에 Stack Guard + Non-executable Stack 패치를 우회하기 위해 [6]에서 제시된 GOT를 덮어쓰는 기술을 사용할 것을 제안하였다. 처음 실행되는 strcpy()에서 p 포인터의 값을 printf() 함수의 GOT Entry 주소로 덮어쓴다. 그리고 strncpy() 함수에서 printf() 함수의 Entry주소를 libc system() 함수의 주소로 덮어쓴다. 이제 printf() 함수가 실행되면 system() 함수가 실행되게 된다. 이 방법은 libc의 주소에 NULL 문자가 포함되도록 한 수정된 Non-executable Stack Patch 상에서는 동작하지 않지만 만약 그림 15와 같은 형태의 취약한 소스가 있다면 우회하는 것이 가능해진다.

```
char global_buf[64];

int f(char *string, char *dst)
{
    char a[64];
```

```
    strcpy(a, string);
    strncpy(dst, a, 64);
}

main(int argc, char **argv)
{
    f(argv[1], global_buf);
    execl("anyprogram", "", 0);
    printf("End of program\n");
}
```

그림 15. 버퍼 오버플로우 취약점을 가지는 예제 코드

아래 그림 16에서 볼 수 있듯이 dst 포인터가 RET 뒤에 위치하므로 이 포인터를 printf() 함수의 GOT로 덮어쓴다. 이후 strncpy() 함수에서 printf() 함수의 GOT주소에 shellcode를 덮어쓰게 되고 printf() 함수가 호출되면 shellcode가 실행되게 된다. Stack Guard, Stack Shield 모두 마지막으로 리턴되기 전에 RET를 체크하기 때문에 이 공격을 탐지할 수 없다.

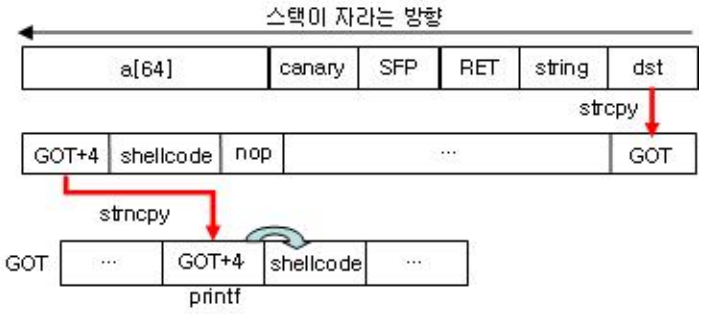

그림 16. Bypass Stack Guard + Non-executable Stack Patch

3.2.3. Getting around strncpy()

[14]가 기고된 Phrack의 동일한 권호에 twitch라는 해커가 그 동안 이론적으로만 알려져 있던 strncpy()를 사용할지라도 프로그램의 형태에 따라 버퍼 오버플로우가 가능함을 증명하는 기사를 발표하였다[15]. 버퍼 오버플로우 취약점이 대표적인 취약점이 된 이후 해당 취약점을 발생시키지 않기 위한 Secure Programming의 방법으로 strcpy() 함수 대신 strncpy() 함수와 같이 복사할 문자의 개수를 같이 입력 받는 함수를 사용할 것이 권고되고 있었다. 하지만 twitch는 많은 프로그래머들이 안전하다고 생각하고 사용하고 있는 strncpy()와 같은 함수들이 자동으로 문자열이나 버퍼를 Null로 종료시키지 않기 때문에 특정 조건 하에서 버퍼 오버플로우가 일어날 수 있음을 지적하였다.

```
char *strncpy(char *dest, const char *src, size_t count);
```

그림 17. strncpy() 함수 원형

strncpy() 함수의 원형은 그림 17과 같으며 src에서 dest로 count만큼 복사한다. 중요한 것은 이 때의 count는 버퍼의 크기이지 문자열의 크기가 아니라는 것이다. 만약 src 문자열의 길이보다 count가 클 경우에는 src 문자열의 Null 문자 이후는 모두 Null 문자로 채워진다. 하지만 src 문자열의 길이가 count보다 크고 src와 dest가 인접해 있다면 버퍼가 Null 문자로 종료되지 않아 버퍼 오버플로우 문제가 생길 수도 있다. 아래 그림 18은 이런 종류의 취약점을 가지는 예제 코드이다.

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

void vul(char *);

int main(char argc, char **argv)
{
    char buf1[1024];
    char buf2[256];

    strncpy(buf1, argv[1], 1024);
    strncpy(buf2, argv[2], 256);

    vul(buf2);
}

vul(char *p)
{
    char buffer[263];
    sprintf(buffer, "%s", p);
}
```

그림 18. strncpy() 함수를 사용해도 버퍼 오버플로우 취약점이 존재하는 예제 코드

아래 그림 19는 sprintf()가 실행되고 난 후의 스택의 모습이다.

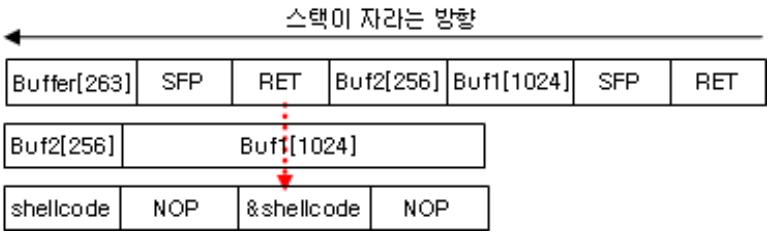

그림 19. Buffer Overflow with strncpy()

vul() 함수에서 p 포인터가 가리키는 buf2는 Null 문자로 끝나지 않기 때문에 sprintf() 함수는 버퍼에 buf1의 Null을 만날 때까지 데이터를 복사하게 되고 공격자는 적절히 offset을 맞춰 RET를 변조할 수 있다.

3.2.4. LibSafe

2000년 6월 스택 오버플로우 공격을 막기 위한 기술중의 하나로 우리에게는 libsafe로 잘 알려진 기술이 USENIX Conference에서 발표된다[16]. 이 기술은 크게 libsafe와 libverify로 구분이 되는데 libsafe는 취약하다고 알려진 모든 함수의 call을 가로채어 안전한 다른 함수로 전환한 뒤 그 결과를 반환한다. 예를 들어 strcpy() 함수가 호출되면 자동으로 src 배열과 dest 배열의 크기를 측정하여 memcpy() 함수를 호출하고 그 결과를 원래의 프로그램으로 반환하는 식이다. 즉 자동으로 버퍼의 크기를 제한하는 것이다. 반면에 libverify는 Stack Guard와 비슷하게 RET의 변조 여부를 체크하지만 Stack Guard와는 달리 프로세스의 메모리를 덮어쓰는 방식을 사용한다.
아래 그림 20과 21은 libsafe와 libverify의 작동방식을 나타낸다.

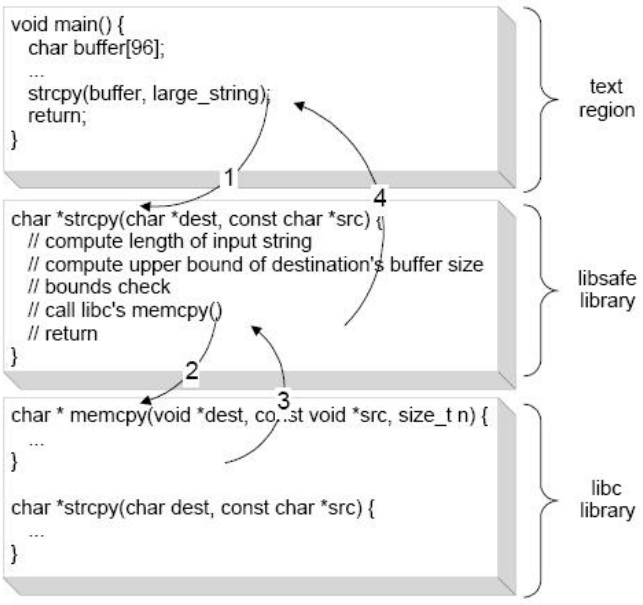

그림 20. LibSafe

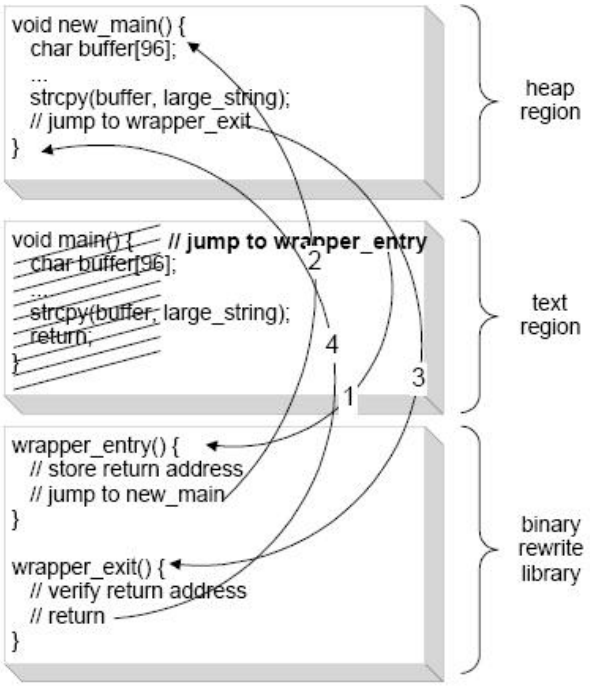

그림 21. LibVerify

3.2.5. Propolice(SSP)

IBM의 연구원인 Hiroaki Etoh가 발표한 컴파일러 기반의 스택 오버플로우 방어 기술로써[17] 후에 Propolice 또는 SSP로 널리 알려지게 된다. 이 기술은 Stack Guard의 XOR Random Canary에 약간의 변형을 가한 것으로써 보호하는 지역이 틀리며 스택 뿐만 아니라 함수 포인터도 함께 보호한다는 특징을 가진다.

아래의 그림 22는 Propolice 기반의 보호된 스택의 모습이다.

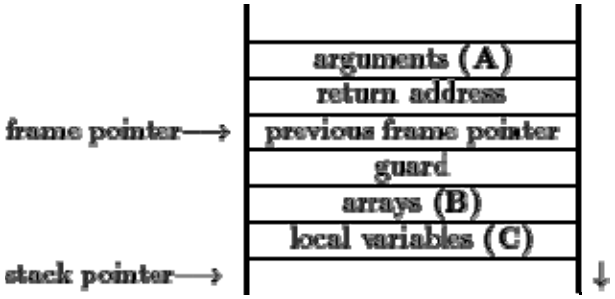

그림 22. Safe Frame Structure

Stack Guard의 Canary와 같은 역할을 하는 guard가 스택의 앞에 주입되는데 초기 버전의 Stack Guard와는 달리 SFP의 앞에 위치하여 RET의 변조 여부를 탐지하게 된다.

아래 그림 23은 Propolice가 Frame Pointer를 보호하기 위해 취약한 소스를 재구성 하는 그림이다. 왼쪽의 취약한 소스가 Propolice에 의해 오른쪽 소스로 변환된다.

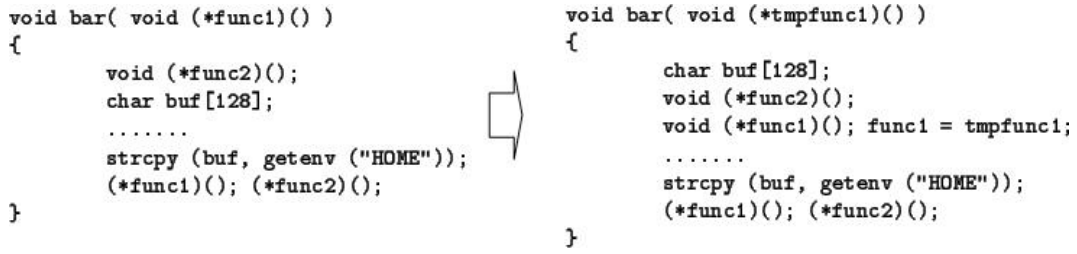

그림 23. Protection form Function Pointer Attack

C 언어의 경우 지역변수는 배치순서에 대한 제약이 없으나 인자들의 배치순서를 바꾸는 것은 허용되지 않는다. 그렇기 때문에 Propolice는 새로운 지역변수를 생성하여 인자인 func1을 가리키게 함으로써 스택 오버플로우에 의한 Frame Pointer의 변조를 탐지한다.

3.2.6. 포맷 스트링

2000년 7월, Pascal Bouchareine가 2000년 6월에 WU-FTP의 취약점으로 SecurityFocus에 발표된 포맷 스트링 취약점[18]을 자세하게 설명하는 기사를 발표한다[19]. 최초의 포맷 스트링 취약점은 1999년 11월에 ProFTPD 1.2.0pre6 버전에서 Tymm Twillman에 의해 발견되었다[22].

이 기술은 전형적인 C 라이브러리들의 취약점을 이용하며 printf() 계열의 함수들을 공격 대상으로 한다. printf() 계열 함수들은 출력 시 포맷 스트링 지정자라고 불리는 인자들을 받아서 데이터의 출력되는 형태를 가공하며 이 포맷 스트링 지정자들은 "%d", "%s", "%l" 등 여러 종류가 있다. 포맷 함수의 인자들은 모두 스택에 저장된다. 포맷 함수는 인자로 전달된 문자를 하나씩 읽으면서 포맷 스트링을 분석하는데 만약 읽어 들인 문자가 '%'가 아니라면 해당 문자는 출력으로 간주되고 '%'일 경우 다음 뒤의 문자를 읽어 인자의 타입을 결정하게 된다. 프로그램이 잘못 작성될 경우 printf() 계열 함수를 통해 임의의 메모리에 대해 read, write가 가능하게 된다.

아래는 포맷 스트링 취약점을 가지는 소스코드이다.

```
void main()
{
    char tmp[512];
    char buf[512];

    while(1) {
        memset(buf, '\0', 512);
        read(0, buf, 512);
        sprintf(tmp, buf);
        printf("%s", tmp);
```

```
    }
}
```

그림 24. 포맷 스트링 취약점을 가지는 예제 코드

그림 24에서 입력 값으로 '%x' 가 들어가게 되면 printf() 함수는 스택의 메모리를 출력하게 된다. 즉 printf() 계열 함수에 최종적으로 들어가는 인자가 아래와 같이 구성될 경우 printf() 함수는 스택의 메모리를 덤프하는 역할을 하게 된다.

```
printf("%x %x %x %x");
```

그림 25. 포맷 스트링 취약점을 이용한 메모리 덤프

만약 특정 주소의 메모리를 덤프하고 싶다면 아래와 같이 입력되도록 하면 된다.

```
printf("\x02\x96\x04\x08[%s]");
```

그림 26. 포맷 스트링 취약점을 이용한 특정 주소의 메모리 덤프

그림 26은 0x08049602부터 Null 바이트에 도달할 때까지 메모리를 덤프한다.
또 printf() 계열 함수들의 포맷 스트링 지정자들 중 %n 이라는 지정자를 이용하면 임의의 메모리에 임의의 값을 쓰는 것이 가능하게 된다. %n 지정자는 출력된 바이트의 수를 반환하는데 아래 그림 27에서 변수 i에는 4가 저장된다.

```
printf("ABCD%n\n", &i);
```

그림 27. printf() 함수의 %n 지정자

임의의 메모리 값을 덮어쓰기 위해서는 아래와 같이 입력되게 하면 된다. 아래 그림 28은 0xbffff7f0 번지에 4를 쓴다.

```
printf("\x70\xf7\xff\xbf%n");
```

그림 28. 포맷 스트링 취약점을 이용한 특정 주소에 쓰기

아래 그림 29는 0xbffff770번지의 값을 0x00000808로 쓴다.

```
printf("\x70\xf7\xff\xbf\x71\xf7\xff\xbf%n%n");
```

그림 29. 포맷 스트링 취약점을 이용한 특정 주소에 특정 값 쓰기

만약 RET가 존재하는 메모리 위치를 알 수 있다면 위의 방법들을 사용하여 RET 값을 변조할 수 있다. 문제는 스택에 주입한 shellcode가 0x80421f와 같은 곳에 존재한다고 가정할 때 RET 주소를 알아도 0x80421f와 같이 큰 값을 입력할 수 없다는 것인데(출력된 값의 바이트 수가 0x80421f = 8405535(10)가 되어야 하므로) 이는 %hn 지정자를 이용하거나 printf() 함수를 여러 번 호출하여 RET 4바이트 값의 하위 번지부터 차례로 1바이트씩 값을 덮어쓰는 방식을 이용하면 RET 값을 shellcode의 주소로 덮어쓰는 것이

가능하게 된다.

3.2.7. Heap Overflow in Netscape Browsers

2000년 7월경 Solar Designer가 “JPEG COM Maker Processing Vulnerability in Netscape Browsers”라는 기사를 발표하는데[24], 이 기사는 JPEG File Format을 읽어들일 때 오버플로우가 발생한다는 것을 이용하였다. 흥미로운 것은 Heap 메모리에서 발생하는 오버플로우를 이용해 malloc() 함수의 내부 구조체를 덮어써서 임의의 코드를 실행하게끔 하는 점이다. 이 기술은 향후 Heap Overflow 공격의 주된 방법으로써 이용되게 된다.

malloc() 함수에 의해 메모리가 할당될 때마다 chunk라고 불리는 구조체가 생성되며 이 구조체들은 double linked list로 연결되어 있다.

아래 그림 30은 chunk 구조체의 내용을 나타낸다.

```
#define INTERNAL_SIZE_T size_t
struct malloc_chunk {
        INTERNAL_SIZE_T prev_size; // Previous chunk의 크기
        INTERNAL_SIZE_T size;
        struct malloc_chunk *fd; // Previous 포인터
        struct malloc_chunk *bk; // Next 포인터
};
```

그림 30. chunk 구조체

할당된 메모리가 해제될 때 chunk는 unlink() 함수에 의해 제거되는데 이 때 만약 이웃한 chunk가 이미 free가 된 상태일 경우 메모리 관리 효율을 위해 병합을 시도한다.

free() 함수 호출 시 실행되는 unlink() 매크로는 아래와 같다.

```
#define unlink(P, BK, FD) {
    BK = P->bk;
    FD = P-fd;
    FD->bk = BK;
    BD->fd = FD;
}
```

그림 31. unlink() 매크로

free()가 실행되면 각 chunk들의 double linked list 구조는 아래와 같이 변하게 된다.

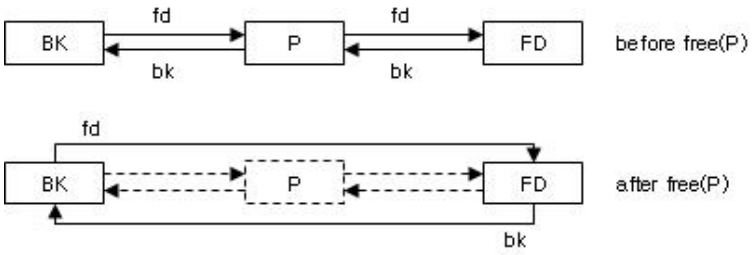

그림 32. free(P)

chunk에는 PREV_INUSE 플래그가 있는데 size의 1바이트 값을 사용하며 이 값이 1일 경우 이전의 chunk는 사용 중임을 나타낸다. 즉 free() 시에 이웃한 chunk의 PREV_INUSE 값이 1이 아닐 경우 병합이 일어나게 된다. 공격자는 오버플로우를 일으켜 next chunk의 13 바이트 즉, 두 개의 포인터(next, previous) 8바이트와 prev_size 4바이트, 그리고 size의 최하위 비트인 PREV_INUSE 플래그를 덮어씀으로써 임의의 코드를 실행하는 것이 가능하게 된다. 다시 free()가 호출되면 unlink()가 실행되고 오버플로우에 의해 만들어진 fake chunk에 의해 shellcode가 실행된다. Solar Designer는 __free_hook을 덮어써서 free()가 다시 호출되도록 할 것을 제안하였다.

3.2.8. PaX

2000년 9월에 grsecurity team[7]에서 x86에서의 Non-executable Page를 구현한 리눅스 패치를 발표하는데[20], 이는 후에 PaX로 널리 알려지게 된다. PaX는 OpenWall Project[3.1.2]와는 달리 스택과 Heap 모두를 보호한다. x86 계열의 CPU는 원래 하드웨어적인 Non-executable Page를 지원하지 않기 때문에 이들은 1996년 7월에 시작된 plex86 project [8] 에서 얻은 아이디어를 사용하여 Non-executable Page를 구현하였다. plex86은 펜티엄 이후에 출시된 x86 기반 CPU에서의 page fault가 일어나는 상태(DTLB, ITLB)에 대해 조사하였는데 PaX는 이 정보를 토대로 Non-executable이 구현된 page에 대해 violate가 일어나지 않는 상태들을 취합하여 읽고 쓸 수는 있지만 실행은 불가능한 페이지를 구현함으로써 프로세스에 주입한 임의의 shellcode를 실행하는 것이 불가능하도록 하였다. 하지만 초기 버전의 PaX는 일반적인 RTL 공격 기술로 우회하는 것이 가능하였다. 현재의 PaX는 PAGEEXEC, SEGMEXEC, Restriction mmap() and mprotect(), ASLR, Random Stack, Random Kernel Stack, Random mmap(), Trampoline emulation, PIC, PIE 등의 기능이 구현되어 있다[41].

3.3. 2001년

1996년부터 시작된 버퍼 오버플로우의 열풍이 2001년에 들어서면서부터는 조금씩

[7] http://www.grsecurity.net/
[8] http://www.plex86.org

가라앉기 시작하는 모습을 보이기 시작한다. 고전적인 스택 오버플로우 보다는 Heap에 대한 연구가 많이 이루어져 Heap 오버플로우 공격 기술이 많이 발표된다.
2000년에 발견된 포맷 스트링 취약점을 공격하는 기술은 버퍼 오버플로우 기술을 대체할 기술로서 연구되며 기존에 발표되었던 기술들과 결합되어 강력한 공격 능력을 가지게 된다.

3.3.1. Overwriting the .dtors section

2001년 3월에 ELF 포맷에서 .dtors 영역을 덮어써서 프로그램의 실행 흐름을 바꿀 수 있음을 증명하는 기사가 발표된다[21]. 이 기술은 gcc 컴파일러에서 지원하는 속성들 중 constructor, destructor을 이용하는데 constructor 속성이 지정된 함수는 main()함수가 시작하기 전에 먼저 실행되고 destructor 속성이 지정된 함수는 main() 함수의 종료 이후에 실행이 된다. constructor는 .ctors 영역으로, destructor는 .dtors 영역으로 표현되는데 기본으로 쓰기 가능한 메모리에 매핑된다. 또한 프로그래머가 특별히 정의하지 않아도 두 영역은 메모리에 매핑된다. ELF 포맷에서 .dtors 영역은 data 영역과 bss 영역 사이에 위치하게 되는데 만약 data 영역에 오버플로우가 난다면 근접한 .dtors 영역을 덮어쓰는 것이 가능하게 된다.
아래 그림 33은 .dtors 영역의 layout을 나타낸다.

```
0xffffffff <function address> <another function address> … 0x00000000
```

그림 33. .dtors 영역의 layout

그림 33의 .dtors 영역의 layout에서 ‘another function address’ 지점에 실행하기 원하는 함수의 주소를 덮어씀으로써 공격자는 shellcode를 실행할 수 있다. 만약 포맷 스트링 공격을 이용하여 .dtors 영역을 덮어쓸 경우 RET를 찾지 않아도 쉽게 shellcode를 실행할 수 있다.
아래 그림 34는 포맷 스트링 취약점을 가지는 예제 코드이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

static void bleh(void);

int main(int argc, char *argv[])
{
    static u_char buf[] = “bleh”;
    if(argc < 2)
        exit(EXIT_FAILURE);
    strcpy(buf, argv[1]);
```

```
    exit(EXIT_SUCCESS);
}

void bleh(void)
{
    printf("goffio!\n");
}
```

그림 34. 포맷 스트링 취약점을 가지는 예제 코드

아래 그림 35처럼 입력 값으로 bleh() 함수의 주소를 .dtors 영역을 덮어씀으로써 정상적으로는 실행할 수 없는 bleh() 함수를 실행시킬 수 있다.

```
$./bleh `perl –e 'print "A" x 24; print "\xb0\x84\x04\x08";`
```

그림 35. bleh() 함수 실행

3.3.2. FormatGuard

2000년 8월에 Stack Guard로 유명한 Crispin Cowan을 비롯한 몇 명이 포맷 스트링 공격을 방어하기 위한 glibc의 패치인 FormatGuard를 발표한다[22].

FormatGuard는 포맷 스트링에 의해 호출된 인자의 개수와 printf()에 제시된 실제 인자의 개수를 비교하는데 만약 실제 인자의 개수가 포맷 스트링 호출에 의한 인자의 개수보다 작다면 FormatGuard는 이를 공격으로 탐지하고 syslog에 공격 시도를 기록한 후 프로그램을 종료한다. 인자의 개수를 알아내기 위해 FormatGuard는 아래의 'Frantzen's Argument Counter' 매크로를 사용한다.

```
#define printf mikes_print(&cnt, print0
#define print0(x, args...) x ,print1(## args)
#define print1(x, args...) x+(++cnt-cnt) ,print2(## args)
#define print2(x, args...) x+(++cnt-cnt) ,print3(## args)
...
void mikes_print(int *args, char *format, ...);
```

그림 36. Argument Counter 매크로

실제 printf()를 보호하기 위한 FormatGuard의 구현인 protected_printf() 함수는 아래와 같다.

```
#define __formatguard_counter(y...)    __formatguard_count1( , ##y)
#define __formatguard_count1(y...)     __formatguard_count2(y,5,4,3,2,1,0)
#define __formatguard_count2(_, x0, x1, x2, x3, x4, n, ys...)  n
#define printf(x...)       __protected_printf(__PRETTY_FUNCTION__,\
```

```
                                        __formatguard_counter(x) -1, ##x)
```

그림 37. protected_printf()

3.3.3. malloc() tricks

Phrack 57호에 malloc()을 기반으로 하는 공격 기술에 관한 주제를 다룬 두 개의 흥미로운 기사가 발표된다. 그 중 하나가 "vudo malloc tricks"[26]라는 기사인데 sudo에 긴 명령어를 입력하여 실행하였을 경우 발생하는 do_syslog() 함수의 취약점을 공격한다. 이 기사에서는 임의의 코드를 실행시키기 위해 두 가지의 공격 기술을 설명한다.

가. unlink() 이용하기

이 기술은 Solar Designer에 의해 고안되었던[3.2.7] 것으로 unlink()시 Linked List를 구성하는 포인터를 조작하여 임의의 코드를 실행하게 하는 것이다. 이 기사에서는 좀 더 자세하게 unlink() 공격 기술을 설명하는데 오버플로우에 의해 덮어 쓰여지는 next chunk는 아래와 구조를 가지게 된다.

그림 38. Fake Chunk

그림 38에서 볼 수 있듯이 prev_size와 size는 Null 바이트를 피하기 위해 0xfffffffc 값으로 덮어쓰는데 주의해야 할 것은 size의 최하위 비트인 PREV_INUSE를 0으로 덮어써야 한다는 것이다.

Solar Designer와는 달리 free()의 GOT를 덮어씀으로써 두 번째 free()가 일어날 때 shellcode를 실행하도록 제안하였다. 그래서 double free() 취약점이라고도 불린다.

나. frontlink() 이용하기

두 번째 공격기술은 frontlink() 매크로를 대상으로 하는데 unlink()를 이용하는 것보다 더 어렵고 또 유연성이 적다.

아래의 그림 39는 frontlink() 매크로를 나타낸다.

```
#define frontlink(A, P, S, IDX, BK, FD) {
    if(S < MAX_SMALLBIN_SIZE) {
        do_anything()..
    }
    else {
        IDX = bin_index(S);
        BK = bin_at(A, IDX);
```

```
        FD = BK->fd;
        if(FD == BK) {
            mark_binblock(A, IDX);
        }
        else {
            while(FD != BK && S < chunksize(FD)) {
                FD = FD->fd;
            }
           BK = FD->bk;
        }
        P-bk = BK;
        P->fd = FD;
        FD->bk = BK->fd = P;
    }
}
```

그림 39. frontlink()

frontlink()에 의해 free()된 chunk P가 MAX_SMALLBIN_SIZE보다 클 경우 while문으로 진입하는데 P가 주입된 지점을 찾을 때까지 진행된다. 만약 공격자가 진행되는 chunk들 중 하나의 forward 포인터를 fake chunk의 주소로 덮어쓸 수 있다면 FD가 이 fake chunk를 가리키게 한 후 루프를 벗어날 수 있다. 다음으로 fake chunk의 back 포인터인 BK가 읽히는데 이 값은 BK로부터 8byte 떨어진 곳에 위치되고 chunk P의 주소로 덮어쓰여지게 된다. 공격자는 fake chunk의 bk 필드 내 함수 포인터의 주소를 저장하고 이 함수 포인터를 chunk P의 주소로 덮어쓰도록 frontlink()를 속일 수 있다. 이 기술은 실제로 구현된 공격 코드가 존재하지는 않지만 특정 상황 하에서 이론상으로는 가능하다.

3.3.4. Once upon a free()

Phrack 57호에 발표된 malloc()을 기반으로 하는 공격 기술에 관한 주제를 다룬 다른 하나의 기사로서 “Once upon a free()”가 발표된다[27]. 이 기사는 [3.3.3]과는 달리 GNU C 라이브러리 기반 malloc()뿐만 아니라 SystemV에서의 malloc()을 기반으로 하는 공격 기술도 같이 다루고 있다. 또 GNU C 라이브러리 기반의 malloc()을 공격할 때 사용할 수 있는 새로운 방법도 제안하였다. 기존의 malloc() 공격 기술이 오버플로우가 일어난 chunk의 next chunk와의 병합을 시도하는 반면에 next chunk의 5바이트만을 덮어씀으로써 이전 chunk와의 병합을 시도하도록 하여 임의의 코드를 실행시킬 수 있음을 증명하였다.

SystemV의 경우 GNU C라이브러리와는 달리 chunk를 splay tree 구조를 이용하여

관리하는데, 그 구조는 아래와 같다.

```
typedef union_w_ {
    size_t      w_i;            // an unsigned int
    struct_t_   *w_p;           // a pointer
    char        w_a[ALIGN]      // to force size
} WORD

typedef struct _t_ {
    WORD t_s; // size of this element
    WORD t_p; // parent node
    WORD t_l;  // left child
    WORD t_r; // right child
    WORD t_n; // next in link list
    WORD t_d; // dummy to reserve space for self_pointer
} TREE
```

그림 40. Splay Tree 구조체

위의 Tree 구조에서 일반적으로 오직 t_s와 t_p 요소만이 사용되고 사용자 data는 t_l부터 저장되며 free()가 아니라 realfree()에서 실질적으로 병합되는 과정이 수행되게 된다. 공격자는 마지막 노드가 아닌 chunk를 오버플로우 시켜 뒤에 fake chunk를 생성하게 되고 만약 malloc() 함수가 다시 호출되거나 free() 함수가 여러 번 실행된다면 결국 두 개의 chunk는 서로 병합되면서 임의의 코드를 실행하게 된다.

3.3.5. Team teso's 포맷 스트링

2001년 9월, 포맷 스트링 취약점에 대한 역사 및 여러 기술들을 종합한 scut의 기사가[31] 17회 Chaos Communication congress를 통해 발표된다. 이 기사에서는 버퍼 오버플로우와 포맷 스트링에 대한 비교 및 2000년에 발견된 주요 포맷 스트링 취약점들에 대해서도 잘 정리하고 있다. 전통적인 포맷 스트링 공격 기술과 더불어 변형된 공격 기술로써 '%hn' 지시자를 이용한 짧은 int 타입 쓰기, '*'을 이용한 Stack popping, '$' 지시자를 이용한 직접적인 인자 접근기술 등을 소개하고 있다. 또 offset 거리를 알아내기 위한 무작위 대입법과 offset을 알 필요 없이 간단히 공격할 수 있는 방법으로써 GOT 덮어쓰기[3.1.4], .dtors 덮어쓰기[3.2.10], __free_hook등의 C라이브러리 hook[3.2.7], atexit 구조체 공격방법[3.2.2], 함수 포인터 덮어쓰기, jmpbuf 덮어쓰기[3.1.8], 포맷 스트링을 이용한 RTL 공격 등을 소개하였다. Heap을 공격하기 위해서는 Heap 메모리에 할당된 버퍼가 스택 상의 다른 버퍼에 영향을 미칠 수 있어야 함을 언급했고 그 밖의 특별한 고려사항 등에 대해서도 짧게 언급하였다.

3.3.6. Advanced return-into-libc

2001년의 마지막 달에 RTL 공격 기술의 레퍼런스로 불려도 손색이 없을 기사[32]가 Phrack에 발표된다. Nergal은 이 문서에서 그 동안 발표되었던 발전된 RTL 기술들과 스택 오버플로우에 대해서 다루고 있다. 기존 RTL의 경우 라이브러리 내의 함수를 실행할 수 있는 기회가 단 한번뿐이라는 제약사항을 극복하기 위해 다양한 방법들이 연구되었는데 esp 값을 올리거나 fake ebp를 사용하는 것들이 대표적이다.

가. esp 값 올리기

gcc 컴파일러에서 –fomit-frame-pointer 옵션을 이용해 컴파일 하면 Frame Pointer가 제거된다. 이 경우 에필로그는 아래와 같은 형식을 가지게 된다.

```
eplg :
        addl    $LOCAL_VARS_SIZE, %esp
        ret
```

그림 41. epilog with –formit-frame-point option

이 때 아래와 같은 형태의 문자열을 통해 여러 개의 함수를 실행할 수 있다.

그림 42. Attack String with LOCAL_VARS_SIZE

그림 42에서 볼 수 있듯이 f1과 f2는 라이브러리 내 함수들의 주소를 나타내며 f1_arg1의 값과 PAD의 길이를 더한 값이 LOCAL_VARS_SIZE 값과 일치해야 한다. 또 함수의 에필로그 부분을 이용하는 것 대신 라이브러리나 프로그램 내에서 아래와 같은 명령이 있는 부분을 찾을 수 있다면 그림 44와 같은 형태의 문자열을 통해 여러 개의 함수를 실행하는 것이 가능하다.

```
pop-ret :
        popl    any_register
        ret
```

그림 43. pop-ret 명령

그림 44. Attack String with pop-ret

하지만 popl을 이용한 기술 역시 3개 이상의 pop을 사용한 형태의 pop-ret을 찾기가 힘들기 때문에 3개 이상의 함수를 실행하는 것이 어렵다.

나. fake ebp 사용하기

다음으로 fake ebp를 사용하는 방법이 있는데 일반적으로 –formit-frame-pointer 옵션 없이 컴파일 된 프로그램을 공격할 때 사용된다. –formit-frame-point 옵션이 없을 경우 프로그램의 에필로그는 아래와 같은 형태를 가지게 된다.

```
leaveret :
        leave
        ret
```

그림 45. epilog without –formit-frame-point option

그림 45와 같은 형태를 가지는 에필로그에 대해 아래와 같은 문자열을 통해 여러 개의 함수를 실행하는 공격을 하는 것이 가능하다.

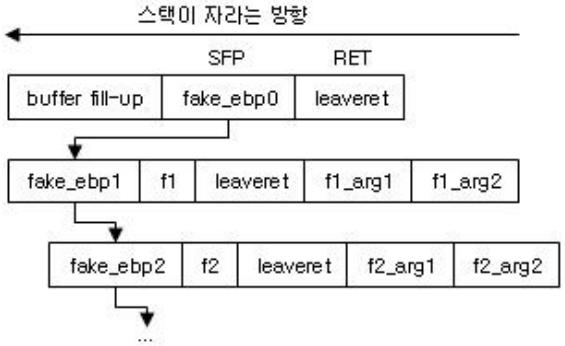

그림 46. Attack String with Fake ebp

그 외 –formit-frame-pointer 옵션이 적용된 프로그램에 fake ebp 기술 사용하는 방법, 권한 상승을 위한 널 바이트 주입 등에 대해 언급하였다.

PaX를 우회하는 방법에 대한 내용은 이 문서에서 가장 흥미로운 부분인데 이 때의 PaX는 초기 버전을 우회하는 공격들을 방어하기 위한 여러 가지 기능이 추가되어 있는 상태였다. 특정 영역의 메모리의 권한을 변경하여 PAGEEXEC를 우회할 수 있는 mmap()과

mprotect() 호출을 제한하기 위한 MPROTECT 패치, 모든 파일 및 임의의 매핑에 대하여 항상 난수화 된 주소 값을 가지게 하여 RTL 공격 시 라이브러리 주소의 획득을 어렵게 하는 mmap() Randomization 등이 그것이다.
하지만 로컬 공격 코드의 경우 라이브러리의 주소와 스택이 위치한 주소가 기록된 /proc/…/maps를 읽을 수 있으므로 공격의 가능성이 존재하고 mmap() 호출이 추측 가능한 범위에서 난수를 발생시킴으로써(Linux Kernel 2.4.18 + grsecurity patch 1.9.4) Brute Force 공격으로 라이브러리의 주소를 획득할 수 있는 가능성이 있음을 지적하였다.
라이브러리가 재배치 될 때 dl-resolve() 함수를 이용한 스택 오버플로우 기술은 PaX를 우회하는 내용 중에서도 가장 흥미로운 부분이다. [3.1.4]에서 살펴보았던 PLT가 호출되는 과정은 아래와 같다.

```
jmp     *some_func_dyn_reloc_entry
push    $reloc_offset
jmp     begging_of_.plt_section
```

그림 47. PLT가 호출되는 과정

.plt 섹션의 시작 부분으로 점프한 후 프로그램의 제어권은 dl_resolve() 함수로 넘어가게 되는데 이 때 reloc_offset도 인자도 같이 넘어가게 된다. 이제 dl_resolve() 함수는 재배치 엔트리, symtab 엔트리 계산과 무결성 검사 등의 과정을 수행한 후 스택 포인터를 조정하고 다른 함수를 호출하게 된다.
만약 공격자가 Elf32_Rel, Elf32_Sym 구조체의 변수를 적당히 조절한다면 dl_resolve() 함수를 통해 다른 함수를 호출할 수 있음을 짧은 코드를 통해 증명하였다. 이 공격에 사용될 문자열은 아래와 같다.

그림 48. Attack String for do_resolve()

성공적인 공격을 위해 미리 Elf32_Rel, Elf32_Sym와 같은 구조체를 적당한 곳에 위치시키는 것이 필요한데, 이를 위해 strcpy(), strncpy(), sprintf()와 같은 함수를 이용할 수 있어야 하며 취약한 프로그램은 정적 변수 또는 malloc()으로 할당된 변수에 데이터를 복사하여야만 한다.

3.3.7. Windows 2000 포맷 스트링

Windows 플랫폼과 Oracle에 대한 취약점 연구로 유명한 David Litchfield가 Windows 2000에서의 포맷 스트링 공격법에 대한 문서[40]를 발표한다. Linux와는 달리 shell을 실행시키기 위해 Winexec() API를 사용한다는 점 외에는 전통적인 포맷 스트링 공격법과

다르지 않다.

3.4. 2002년 ~ 2004년

2001년 후반부터는 Hacker들의 공격 목표가 Linux 중심에서 Windows 중심으로 조금씩 방향이 바뀌는 특이한 공격 흐름이 생성된다. 소스가 공개되어 있는 리눅스와는 달리 비교적 안전한 대상으로 여겨졌던 Windows에 대한 연구가 조금씩 이루어지면서, 여러 방어 기술들이 운영체제 수준에서 구현된 리눅스보다 상대적으로 손쉬운 Windows가 주 공격 대상이 된다. 리눅스의 경우는 Exec-Shield가 주로 공격 대상이 되었고 Windows의 경우는 XP Service Pack 2부터 적용된 여러 방어 기술들이 공격 대상이 된다.

3.4.1. Bypassing PaX ASLR protection

Phrack 59호에 또 다시 PaX를 우회할 수 있는 공격법이 발표된다[33]. 이 때의 PaX는 [3.3.6]에 구현된 기술들 외에 ET_EXEC가 추가되었다. 이는 [3.3.6]에서 제시된 RTL 변형 공격 기술을 방어하기 위해 추가된 기술로써 ET_DYN 오브젝트 안의 ET_EXEC ELF 오브젝트를 재 링크시켜 프로그램 실행 시마다 main 오브젝트의 주소가 비 고정적인 주소를 가지도록 한 기술이다.

INTEL Pentium 보호모드의 페이지 시스템에서 0804812C라는 주소는 아래와 같이 구성된다.

08 : 페이지 디렉토리 엔트리 인덱스
048 : 페이지 테이블 엔트리 인덱스
12C : 페이지 offset

그림 49. INTEL Pentium 보호모드 페이지 시스템에서의 주소 구성

그림 49에서 볼 수 있듯이 상위 20비트가 같은 두 주소는 같은 페이지 내에 존재함을 나타낸다. Tyler는 PaX의 ASLR은 각 라이브러리의 주소가 프로그램 실행 시마다 바뀌어 비 고정적인 값을 가지지만 각 함수들은 동일한 페이지 내에 존재하기 때문에 하나의 함수 주소를 알아낼 수 있다면 offset을 계산하여 다른 함수들의 주소를 알아낼 수 있음을 지적하였다. 즉, EIP의 하나 또는 두 개 바이트만 오버플로우 시켜도 동일한 페이지 내의 다른 함수로 돌아가는 것이 가능해지는 것이다. 만약 취약한 부분이 스택 상에 있고 printf() 함수를 가지고 있어서 전달되는 인자를 조작할 수 있다면 포맷 스트링 취약점을 이용해 RTL 공격을 성공시킬 수 있게 된다. 공격자가 __libc_start_main()으로부터 main()의 RET와 가상주소를 알아내었다면 라이브러리의 각 함수들은 아래와 같은 계산을 통해 주소를 알아낼 수 있다.

함수의 주소 = main()의 RET + (함수의 offset – main()의 RET의 offset)

그림 50. offset을 이용한 함수들의 주소 계산

공격 문자열에 사용할 함수들의 인자들을 적당히 배치시킨 다음 함수를 실행시키기 위해 %esp lifting 기술을 사용한다. 이를 위해 라이브러리의 .text 영역에서 pop-ret 또는 pop-pop-ret 명령어 집합이 위치한 주소를 알아내는 작업이 추가로 필요하다. Tyler는 이런 방식을 사용하면 RET 뒤로 단지 84바이트만을 공격 문자열로 덮어씀으로써 RTL 공격을 통해 PaX의 ASLR을 우회할 수 있음을 증명하였다.

3.4.2. Smashing the Heap under Win2k

BlackHat USA 2002에서 Halvar Flake라는 해커가 제 3세대 공격 기술로 포맷 스트링 공격과 Heap Overflow를 지목하였는데 그 중 Windows 2000 환경 하에서의 Heap Overflow에 대해 발표[50]를 한다. 그는 리버스 엔지니어링과 Undocumented NT[9]를 통해 windows 2000의 Heap 메모리 관리 알고리즘을 분석하고 오버플로우를 통해 임의의 코드를 실행하는 것이 가능하다는 것을 증명해 보였다. C 런타임 라이브러리인 malloc()과는 달리 Windows에서는 Heap 메모리를 할당 받기 위해 아래 그림 51, 52와 같이 세 가지 방법을 이용할 수 있다.

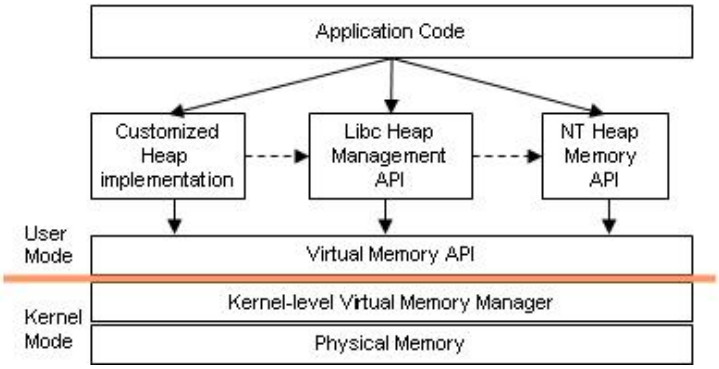

그림 51. Win32 Heap Model

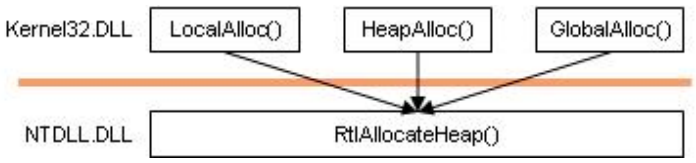

그림 52. Win2k Heap API

User Mode에서의 LocalAlloc(), HeapAlloc(), GlobalAlloc() 호출은 Kernel Mode에서 Native API인 RtlAllocateHeap()을 호출하게 된다. C 런타임 라이브러리와 동일하게 chunk는 header와 data로 구성되고 각 chunk는 Linked List로 관리된다. Windows 2000에서의 Heap Overflow 역시 할당된 Heap 메모리가 free()될 때 Flink와 Blink 포인터를 조작하여 임의의 코드를 실행하는 방법을 사용한다.

[9] http://undocumented.ntinternals.net/

3.4.3. Integer Overflow

2002년 12월에 버퍼 오버플로우로 연결되는 약간 특이한 형태의 Overflow 취약점이 발표된다[37]. Integer Overflow로 명명된 이 취약점은 숫자 형으로 선언된 변수가 가질 수 있는 값의 범위를 넘어서는 값이 저장될 때 발생하며 이를 통해 특정 상황 하에서 버퍼 오버플로우를 야기시킬 수 있다. Integer Overflow는 직접적으로 메모리를 덮어쓰거나 실행 흐름을 통제하는 방식이 아니기 때문에 탐지가 힘들고 방어하기가 어렵지만 공격하기도 어렵다는 특징을 가지고 있다. 이 기사에서 blexim은 Integer Overflow의 몇몇 형태를 예로 들었으며 각각의 예제는 아래와 같다.

가. widthness bug

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

int main(int argc, char *argv[])
{
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3)
        return -1;

    i = atoi(argv[1]);
    s = i;

    if(s >= 80)
        return -1;
printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

return 0;
}
```

그림 53. widthness bug 예제 코드

위 프로그램의 argv[1]에 int형의 범위를 넘어가는 65536이 입력될 경우 s값에 대한 경계 값 체크를 우회하게 되고 memcpy()에 의해 버퍼 오버플로우가 일어나게 된다.

나. Arithmetic Overflow

```
int catvars(char *buf1, char *buf2, unsigned int len1, unsigned int len2)
{
    char mybuf[256];
    if( (len1+len2) > 256 )
        return -1;

    memcpy(mybuf, buf1, len1);
    memcpy(mybuf+len1, buf2, len2);

    do_something(mybuf);

    return 0;
}
```

그림 54. Arithmetic Overflow 예제 코드

위 프로그램에 len1 = 0x104, len2 = 0xfffffffc 의 값이 입력될 경우 len1+len2의 값이 0x100이 되어 if문을 우회하게 되고, memcpy()에 의해 버퍼 오버플로우가 일어나게 된다.

다. Signedness Bug

```
int copy_something(char *buf, int len)
{
    char kbuf[800];

    if(len > sizeof(kbuf))
        return -1;

    return memcpy(kbuf, buf, len);
}
```

그림 55. Signedness Bug

위 프로그램에서 memcpy()는 len의 값을 unsigned int형으로 받아들이지만 len은 int형이므로 만약 len의 값이 음수라면 if문을 우회하게 되고 memcpy()에서 len이 unsigned int형 값으로 변환되면서 kbuf에 버퍼 오버플로우가 일어나게 된다.

3.4.4. Detect Integer Overflow

흥미롭게도 [3.4.3]이 게재된 Phrack의 동일한 권호에 Integer Overflow를 탐지할 수 있는 기술에 대한 기사가 같이 게재된다[66]. 코드 안에서 입력 값으로 받은 숫자 값이 명확하지 않게 사용되고, 또 입력 값으로 받은 숫자 값이 카운터로 사용되는 반복 루프가 존재할 때 Integer Overflow 취약점이 존재하게 되는 것에 주목하여 만약 반복 루프가 실행되기 전에 루프의 경계 값을 검사한다면 Integer Overflow 취약점을 탐지할 수 있음을 제안하였다. 그리고 이를 증명하기 위해 gcc 컴파일러의 확장 패치의 형태로 구현하였다.

3.4.5. Exec Shield

Redhat에서 Linux에 대한 자동화 공격이나 worm에 대한 방어를 위해 시작된 프로젝트가 2003년 Fedora Core Linux 1과 Enterprise Linux 3에 포함되면서[42] Linux 설치 후 시스템 관리자가 Kernel Patch를 통해 수동으로 보안 수준을 향상시키던 방식에서 벗어나게 되었다. Redhat은 Redhat Linux 7부터 본격적으로 버퍼 오버플로우 방어기술을 운영체제 수준에서 구현하기 시작하였는데 Fedora Core Linux부터는 Exec Shield로 통합되어 구현되게 되었다. PaX와 마찬가지로 Exec Shield 역시 새로운 Fedora Core Linux 버전이 출시될 때 마다 새로운 기능이 추가되었는데 Non-executable Stack, Non-executable Heap, 16MB 미만의 주소를 가지는 Library, Random Stack, Random Library, PIE Compile등의 기능을 포함하게 된다. 비슷한 시기에 구현된 W^X 역시 Exec Shield와 마찬가지로 PaX와 비슷한 기능을 추가하여 OpenBSD에 포함되었다[58].

3.4.6. Windows 2003 Stack Overflow 방지 기술 우회

Microsoft는 스택 오버플로우를 방지하기 위해 Visual Studio에 Stack Guard와 비슷한 방법의 방어 기술을 추가하였다. 프로그램 컴파일 시에 컴파일러는 RET 앞에 XOR Random Canary와 동일한 역할을 하는 Security Cookie(이하 SC)를 추가하여 버퍼 오버플로우 공격을 탐지하게 된다. David Litchfield는 이 보호 매커니즘을 우회할 수 있는 여러 가지 가능성에 대해 언급하였다[46].

가. Stack Overflow를 통한 SEH 덮어쓰기

SEH는 Structured Exception Handling의 약자로서 Windows에서 지원하는 예외처리 기술이다. 프로그램의 실행 중 예외 상황이 발생했을 때 SEH에 의해 해당 예외가 처리되게 된다. Exception 자체는 윈도우에서 이벤트의 하나로 정의되어 있으며 예외 발생 시 해당 예외에 맞는 적당한 handler가 실행되게 된다. 예외처리 핸들러의 구조는 아래와 같이

Linked list로 되어 있으며 스택 내에 존재하기 때문에 오버플로우를 통해 handler 포인터를 덮어써서 임의의 코드를 실행할 수 있다.

```
typedef struct _EXCEPTION_REGISTRATION
{
    EXCEPTION_REGISTRATION *prev;
    EXCP_HANDLER handler;
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

그림 56. Exception Handler 구조체

SEH를 덮어써서 handler를 실행시키기 위해서는 몇 가지 제약이 따르는데 성공적인 실행을 위한 시나리오는 아래의 3가지로 요약된다.

① handler를 가리키는 포인터를 등록된 handler를 가리키게 설정한다.
② handler를 가리키는 포인터를 로드된 모듈의 범위 바깥에 있는 주소를 가리키게 한다.
③ handler를 가리키는 포인터를 Heap 주소를 가리키게 한다.

각 시나리오 별 자세한 사항은 [45]를 참고하도록 한다.

나. SC 덮어쓰기

SC 덮어쓰기는 .data 영역에 저장된 SC의 값을 예측 가능하거나 일정할 경우 스택에 저장된 SC를 . data 영역에 저장된 SC의 값으로 덮어씀으로써 SC 비교를 우회하는 방법이지만 실제 SC는 실행시간마다 난수화 되어 생성되기 때문에 그 값을 예측하는 것이 불가능하다.

다. Security Handler 덮어쓰기

프로그램 실행 시 SC가 일치하지 않을 경우 Security Handler가 정의되어 있는지를 살펴보는데 만약 Security Handler가 정의되어 있다면 호출을 한다. Security Handler 역시 쓰기 가능한 .data 영역에 저장되기 때문에 Security Handler를 덮어쓰거나 새로 설정하여 임의의 코드를 실행시킬 수 있다.

라. Windows System 디렉토리 교체하기

Security Handler가 정의되어 있지 않을 경우 UnhandledExceptionFilter() 함수가 호출되는데 이 함수는 다시 GetSystemDirectoryW() 함수를 호출하여 디렉토리 정보를 얻는다. 디렉토리 정보는 포인터로 반환되어 .data 영역에 저장되게 되고 이 포인터를 덮어써서 임의의 코드를 실행할 수 있게 된다.

마. Ldr Shim 함수 포인터 덮어쓰기

UnhandledExceptionFilter() 함수가 호출하는 LoadLibraryExW()와 FreeLibrary() 함수

포인터를 덮어쓸 수 있다.

Microsoft는 SEH Overwrite 공격법을 방어하기 위해 Visual Studio 2005부터 /SafeSEH라는 옵션을 추가하여 프로그램을 컴파일 하도록 권고하고 있다.

3.4.7. Windows Heap Overflow

2004년 BlackHat USA에서 David Litchfield는 여러 가지 발전된 Windows Heap Overflow 공격 기술에 대해 발표한다[52]. 이 발표에서 David Litchfield는 Heap을 오버플로우 시킨 후 Access Violation이 일어나는 것을 피하기 위해 TotalFreeSize 플래그를 0x14로 덮어쓴 후 첫 번째 FreeList의 Flink와 Blink를 fake control 구조체를 가리키게 할 것을 제안하였다. 또 정의되지 않은 예외(Unhandled Exception)를 발생시켜 이 때 실행되는 UEF(Unhandled Exception Filter) 함수를 덮어씀으로써 임의의 코드를 실행할 수 있음을 증명하였다.

Windows XP에서 새로 추가된 VEH(Vectored Exception Handler)는 SEH(Structured Exception handler)와는 달리 Heap에 저장되며 AddVectoredExceptionHandler(), RemoveVectoredExceptionHandler() 함수를 사용했을 때 호출된다. VEH의 구조는 아래와 같다.

```
struct _VECTORED_EXCEPTION_NODE
{
        DWORD    m_pNextNode;
        DWORD    m_pPreviousNode;
        PVOID    m_pfnVectoredHandler;
}
```

그림 57. VEH 구조체

VEH는 어떤 프레임 기반 핸들러보다도 가장 먼저 호출된다. 그러므로 공격자는 첫 번째 VECTORED_EXECPTION_NODE를 덮어씀으로써 임의의 코드를 실행하는 것이 가능하다. 각각의 프로세스는 PEB(Process Environment Block) 구조체를 포함하고 있으며 TEB(Thread Environment Block)로부터 참조된다. 이 TEB는 FS[0] 레지스터를 통해 접근이 가능하다. PEB는 RtlEnterCriticalSection과 RtlLeaveCriticalSection를 가리키는 포인터를 가지고 있으며 PEB의 위치는 Windows NT / 2000 / XP에서 모두 동일하기 때문에 RtlEnterCriticalSection 위치(0x7FFDF020) 역시 동일하다. 그러므로 오버플로우를 통해 PEB 안의 RtlEnterCriticalSection을 가리키는 포인터를 덮어써서 임의의 코드를 실행하는 것이 가능하다. 또 TEB가 가리키는 첫 번째 Exception Handler를 가리키는 포인터를 덮어쓰는 방법도 제안하였다. Heap 메모리가 Non-executable로 설정되어 있을지라도 [3.4.6]에서 언급한 UEF의 GetSystemDirectoryW() 함수를

이용하여 임의의 코드를 실행할 수 있게 된다.

3.4.8. Windows XP Service Pack 2

Microsoft는 Blaster worm으로 인한 Windows XP 피해를 방지하기 위해 Service Pack 2에 여러 방어 기술들을 포함하는데 그 중 메모리 보호를 위해 Execution Protection(NX) 기능과 Sandboxing 기능을 추가하였다[48]. Execution Protection 기능은 CPU가 제공하는 하드웨어 기능을 이용하며 가상 메모리 페이지 별로 작동한다. 그림 58에서 볼 수 있듯이 보통 PTE의 비트를 변경하여 메모리 페이지에 표시한다.

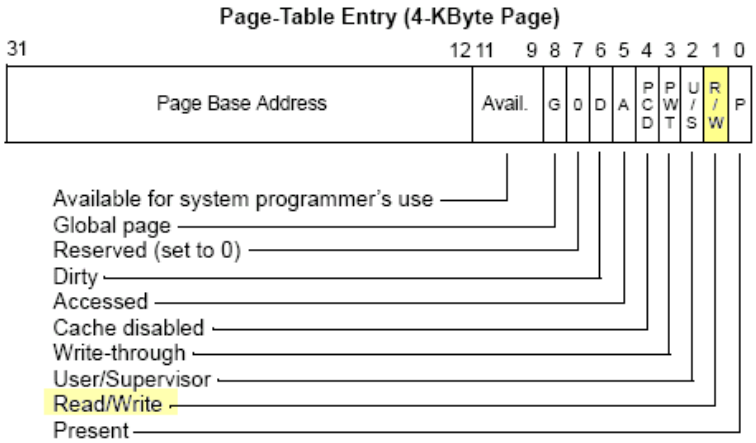

그림 58. Page Table Entry

Execution Protection 기능 외에 Stack과 Heap에 대한 보호도 제공하는데 이를 Sandboxing이라고 한다. Stack 보호를 위해 모든 바이너리는 [3.4.6]에서 언급한 /GS 옵션을 이용하여 재 컴파일 되었고 Heap 할당 시에도 Security Cookie가 추가되도록 변경되었다. 즉 chunk header에 Security Cookie의 무결성을 확인하는 코드와 메모리 해제 시 Flink와 Blink의 값을 확인하는 Safe Unlinking, PEB Randomization 등의 기능이 추가되어 [3.4.2]에서 언급된 방식의 오버플로우를 방지할 수 있다.

3.4.9. Bypass Exec Shield on the Fedora Core Linux 2

Redhat이 Exec Shield 전문[42]에서도 밝혔듯이 Exec Shield가 버퍼 오버플로우 공격에 대해 많은 기능을 구현하였지만 모든 종류의 공격을 다 막을 수 있는 것은 아니다. 국내의 Beist Security Research Group에서 Exec Shield가 적용된 Fedora Core Linux 2에서 버퍼 오버플로우 공격이 가능함을 증명하는 기사[43]를 발표하였다. Exec Shield 하에서 버퍼 오버플로우 공격을 성공시키기 위해 Program Data Segment 영역을 이용하였는데, Program Data Segment 영역은 매핑되는 위치가 항상 고정되어 있고 읽기/쓰기가 가능하며 16MB 이상의 값을 주소로 가지고 있다. 공격자는 ebp를 보호하기 위해서 RTL 공격기술을 이용해 execl()+3 지점으로 접근한다. 그리고 Program Data Segment 영역

내의 적절한 지점을 execl()의 인자 값으로 사용한다. 이 때 execl()에서 실행할 파일 이름을 나타내는 첫 번째 인자 값은 25바이트 길이의 데이터를 가지고 있는데 이 데이터 이름을 가진 심볼릭 링크를 걸어둔다. execl() 함수의 경우 16MB 이하의 값을 주소로 가지지만 마지막 바이트를 입력하지 않는 기술을 이용해 단 한번 사용하는 것이 가능하므로 버퍼 오버플로우 공격을 성공시킬 수 있다.

3.4.10. Heap Spray

최초의 Heap Spray라는 기술은 2001년 eEye Digital Security에서 발표한 권고문에서 알려졌는데[53] 2004년 Skylined라는 해커가 MS04-040 [10] 으로 알려진 Internet Explorer(이하 IE)의 취약점을 발표하면서 널리 알려지게 된다. Skylined는 IE에 오버플로우가 발생하여 비정상적인 메모리로 jmp 또는 call할 때 이 영역에 javascript를 이용하여 Heap을 생성한 뒤 shellcode를 위치시킴으로써 임의의 코드를 실행하도록 하였다. 아래 그림 22에서 볼 수 있듯이 IE가 jmp 또는 call하는 비정상적인 메모리 영역이 정상적인 메모리가 될 때까지 shellcode를 포함하는 Heap 메모리를 계속 주입하기 때문에 이 기술이 Heap Spray로 불리게 되었다.

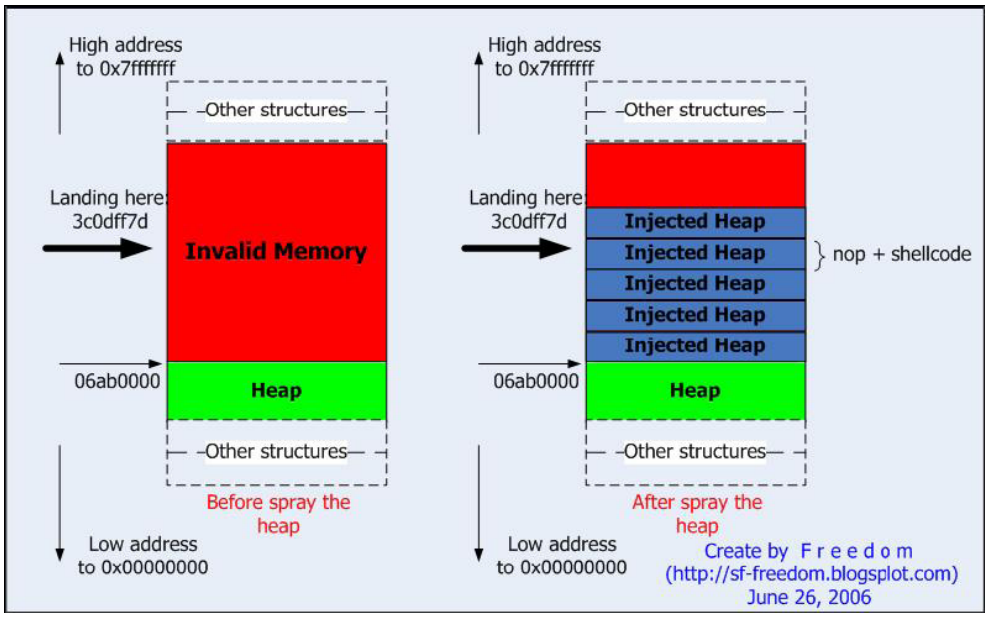

그림 59. Heap Spray Attack

3.5. 2005년 이후

2005년에 들어서면서 본격적으로 Windows에 대한 공격이 집중되기 시작하는데 이는 Windows 2000때부터 시작되었던 내부 구조에 대한 연구가 많이 이루어졌기 때문이다. 여기에는 2004년에 Windows 2000의 소스코드가 유출된 사건도 한 몫을 하게 된다. 리버스 엔지니어링 기술이 발달하면서 심지어는 Windows의 클론을 만들어보고자 하는

[10] http://www.microsoft.com/technet/security/bulletin/ms04-040.mspx

오픈 소스 프로젝트도 시작되는데 ReactOS[11]라고 불리는 이 운영체제는 Windows XP와 거의 동일한 인터페이스를 구현하는데 성공한다. 이는 Windows 내부 구조를 거의 완벽하게 재구성 했음을 의미하는 것이다. 이로써 Windows에 대한 공격 기술의 난이도 역시 점점 더 높아지게 된다. 또한 온라인 게임 계정 탈취 등의 금전적 이익을 위해 공격의 대상이 Client로 바뀌게 되면서 시스템 의존적인 취약점보다는 IE나 Office 관련 취약점들이 주로 연구되게 되는데 주로 ActiveX, Outlook, Excel이 공격 대상이 된다.

3.5.1 Evasion DEP with VirtualAlloc()

VirtualAlloc() 함수는 DEP를 우회하는 가장 첫 번째 아이디어로 제안되었으며[59] 이는 PaX 초창기에 PaX를 우회하기 위해 mprotect() 함수를 이용하던 기술과 비슷하다. VirtualAlloc() 함수는 할당되는 메모리의 권한을 설정해 줄 수 있으므로 RTL 공격 기술을 이용하여 VirtualAlloc()을 호출한 뒤 메모리를 할당하고 실행권한을 설정한다. 이후 memcpy()에 진입하여 shellcode를 할당된 메모리에 복사하고 shellcode를 실행함으로써 DEP를 우회할 수 있다. 이와 비슷하게 VirtualProtect() 함수를 이용하여 페이지의 속성을 실행 가능하도록 변경하여 DEP를 우회할 수도 있다. 최근에는 RootKit에서 Dll Injection을 위한 기술로 VirtualAlloc(), VirtualProtect()가 사용되고 있다.

3.5.2 Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass

DEP(Data Execution Prevention)는 Windows XP Service Pack2에 포함된 메모리 보호 기능 중 Execution Protection을 사용하는 기능이다. DEP는 크게 하드웨어 기반 DEP와 소프트웨어 기반 DEP로 나눌 수 있는데, NX 또는 EVP 기능이 표시된 CPU를 사용할 경우 하드웨어 기반 DEP가 동작하고 그렇지 않을 경우 소프트웨어 기반 DEP가 동작하여 Stack과 Heap를 보호한다. [3.4.8]에서 살펴보았듯이 chunk 헤더에 Cookie를 추가하고 Freelist에 대한 Safe Unlinking 기능을 추가함으로써 기존의 Freelist를 이용한 Heap Overflow 공격 기술은 더 이상 사용할 수 없었다.

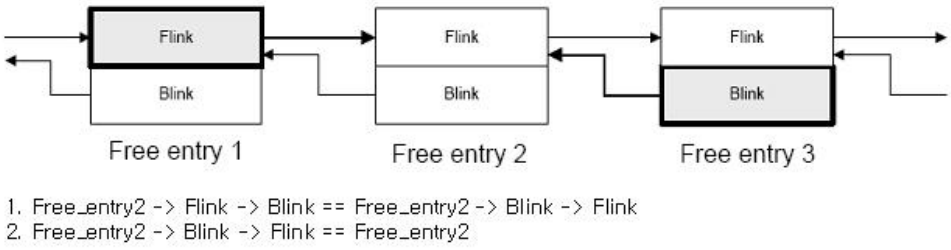

그림 60. Safe Unlinking

하지만 할당되어야 할 메모리가 1016바이트보다 작을 경우 속도 향상을 위해 Windows

[11] http://www.reactos.org/en/index.html

Heap Manager는 single linked list인 lookaside list를 생성하여 메모리들을 관리한다. 이 경우 생성된 Heap 메모리들의 할당/해제 시에는 lookaside list를 확인하게 되는데 이 lookaside list에 대해서는 어떤 검사도 하지 않는다. 그러므로 1016 바이트 보다 작은 메모리 영역을 이용하여 lookaside list의 Flink 포인터를 임의의 값으로 수정할 수 있으므로 임의의 코드를 실행하는 것이 가능하게 되어 DEP을 우회할 수 있게 된다[56].

3.5.3 Bypassing Windows Hardware-enforced Data Execution Prevention

[3.5.2]이 소프트웨어 기반 DEP를 우회한 것과는 달리 리버스 엔지니어링을 통해 하드웨어 기반 DEP를 분석하여 이를 우회하는 기술이 Uniformed 저널 [12] 에 발표되었다[57].

하드웨어 기반 DEP는 CPU가 지원하는 NX 비트를 페이지에 표시함으로써 구현된다. Microsoft는 많은 third-party 제품들과의 호환성을 위해 DEP를 4가지 정책에 따라 작동하도록 구분하였는데 이는 아래와 같다.

가. OptIn : 중요한 시스템 실행파일과 사용자가 추가한 실행파일에 대해서만 동작함.
나. OptOut : 사용자가 추가한 프로그램을 제외한 모든 프로세스에 대해 동작함.
다. AlwaysOn : 전체 시스템에 대해 동작하며 예외목록을 사용할 수 없음.
라. AlwaysOff : DEP를 사용하지 않음

그림 61. DEP 정책의 종류

이 중 OptIn은 Windows XP Service Pack2의 기본 옵션으로 설정되어 있고 Windows 2003에서는 OptOut이 기본 옵션으로 설정되어 있다. DEP의 사용 여부는 프로세스 단위로 실행시간에 결정되며 이를 위해 LdrpCheckNXCompatibility가 ntdll에 추가되었다. dll이 LdrpRunInitializationRoutines를 통해 프로세스의 Context에서 로드 될 때마다 LdrpCheckNXCompatibility가 호출되는데 가장 먼저 SafeDisc DLL이 로드되어 있는지를 살펴본다. 만약 로드되어 있다면 프로세스가 NX 기능을 사용하지 않을 것으로 표시된다. 두 번째로 프로세스를 사용자가 지정한 프로그램 리스트에 등록되어있는지를 살펴보고 마지막으로 dll이 NX 기능과 호환되지 않는 섹션을 가지고 있는지의 여부를 살펴본다.이 세가지 검사 후에 프로세스에 대한 NX 기능의 사용 여부가 새로운 PROCESSINFOCLASS 구조체의 ProcessExecuteFlags를 설정함으로써 결정된다. NtSetInformationProcess()가 이 구조체와 함께 호출되었을 때 4바이트의 bitmask가 buffer의 인자로써 MmSetExecuteOption으로 전달된다. 이 값이 0x1일 경우 NX가 활성화되고 0x2일 경우 NX는 비활성화 되며 비 실행 메모리 지역에서 코드가 실행될 때 이 값을 참고하여 실행 허가가 결정되게 된다.

NtSetInformationProcess(

[12] http://www.uninformed.org

```
        NtCurrentProcess(),
        ProcessExecuteFlags,
        &ExecuteFlags,
        Sizeof(ExecuteFlags));
```

그림 62. NtSetInformationProcess()

만약 공격자가 RTL 공격 기술을 사용하여 NtSetInformationProcess() 함수의 ExecuteFlags 변수를 0x2로 덮어쓸 수 있다면 비 실행 메모리에서 임의의 코드를 실행할 수 있게 되지만 공격 과정에서 NULL 바이트가 반드시 필요하기 때문에 이 기술의 사용은 불가능하다.

skype와 Skywing은 하드웨어 기반 DEP를 우회하기 위해서 return-into-plt 기술과 비슷하게 프로세스에 대한 NX 기능을 비활성화하도록 설정된 ntdll의 코드를 이용할 것을 제안하였다. 먼저 취약한 함수의 RET을 NtdllOkayToLockRoutine로 설정한다.

```
ntdll!NtdllOkayToLockRoutine:
7c952080  b001            mov     al, 0x1
7c952082  c20400          ret     0x4
```

그림 63. NtdllOkayToLockRoutine

NtdllOkayToLockRoutine() 함수는 al에 1을 설정하고 리턴되는 간단한 함수다. 이 리턴 값을 LdrpCheckNXCompatibility+0x13 지점으로 설정한다.

```
ntdll!LdrpCheckNXCompatibility+0x13:
7c91d3f8  3c01            cmp     al, 0x1
7c91d3fa  6a02            push    0x2
7c91d3fc  5e              pop     esi
7c91d3fd  0f84b72a0200    je      ntdll!LdrpCheckNXCompatibility+0x1a
(7c93feba)
```

그림 64. LdrpCheckNXCompatibility+0x13

LdrpCheckNXCompatibility+0x13 지점에서는 esi 값을 2로 설정하고 al 값이 1이면 0x7c93feba로 점프한다.

```
ntdll!LdrpCheckNXCompatibility+0x1a:
7c93feba  8975fc          mov     [ebp-0x4], esi
7c93febd  e941d5fdff      jmp     ntdll!LdrpCheckNXCompatibility+0x1d
(7c91d403)
```

그림 65. LdrpCheckNXCompatibility+0x1a

0x7c93feba 지점에서는 ebp-0x4에 2를 설정하고 0x7c91d403으로 점프한다.

```
ntdll!LdrpCheckNXCompatibility+0x1d:
7c91d403  837dfc00          cmp    dword ptr [ebp-0x4], 0x0
7c91d407  0f8560890100      jne    ntdll!LdrpCheckNXCompatibility+0x4d
(7c935d6d)
```

그림 66. LdrpCheckNXCompatibility+0x1d

0x7c91d403 지점에서는 ebp-0x4의 값이 0이 아니면 0x7c935d6d로 점프한다.

```
ntdll!LdrpCheckNXCompatibility+0x4d:
7c935d6d  6a04              push   0x4             ; Length =  4
7c935d6f  8d45fc            lea    eax, [ebp-0x4]
7c935d72  50                push   eax             ; &ExecuteFlags
7c935d73  6a22              push   0x22            ; ProcessExecuteFlags
7c935d75  6aff              push   0xff            ; NtCurrentProcess()
7c935d77  e8b188fdff        call   ntdll!ZwSetInformationProcess (7c90e62d)
7c935d7c  e9c076feff        jmp    ntdll!LdrpCheckNXCompatibility+0x5c
(7c91d441)
```

그림 67. LdrpCheckNXCompatibility+0x4d

0x7c935d6d 지점에서는 ZwSetInformationProcess()를 호출하기 전 인수들을 스택에 집어넣는데 &ExecuteFlags 부분에 0x2가 설정되는 것을 볼 수 있다. 이제 NX 기능은 비활성화 되고 0x7c91d441 지점으로 점프한다.

```
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91d441  5e                pop esi
7c91d442  c9                leave
7c91d443  c20400            ret 0x4
```

그림 68. LdrpCheckNXCompatibility+0x5c

0x7c91d441 지점에서는 epilogue에 들어가는데 이 때 RET을 shellcode가 저장된 주소로 지정하여 코드를 실행할 수 있다.

3.5.4 Buffer Underrun

2005년 9월에 David Litchfield가 Buffer Underrun이라는 기술을 소개한다[60]. 이

기술은 [3.4.3]에서 살펴본 Integer Overflow와 비슷한 사례를 이용하여 공격한다.

```
#include <windows.h>
int foo(char *str);
int main(int argc, char *argv[]) {
foo(argv[1]);
return 0;
}

int foo(char *str) {
int padding = 0;
int i=0;
char *p=NULL;
char buffer[33]="";

padding = 32 - strlen(str);
for( i = 0; i < padding; ++i )
buffer[i] = '0';

p = &buffer[ padding ];
lstrcpy( p , str );
printf("%s\n",buffer);

return 0;
}
```

그림 69. Buffer Underrun 취약점을 가지는 예제 코드

만약 foo() 함수에 전달되는 인자 str의 크기가 32보다 크다면 padding 변수는 음수를 가지게 되고 Buffer Underrun이 일어나게 된다.

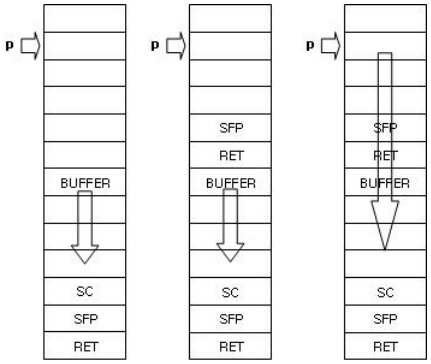

그림 70. Buffer Underrun Attack

그림 70은 왼쪽에서부터 각각 lstrcpy() 호출 전, lstrcpy() 진입 시, lstrcpy() 호출 후의 스택의 모습을 나타낸다. 그림 23에서 알 수 있듯이 소프트웨어 기반 DEP는 Buffer Underrun을 방지할 수 없다.

3.5.5 Bypass Windows Heap Protection

[3.5.2]에 이어 Window Heap 보호기술을 우회하는 새로운 기술에 대한 기사가 발표된다[61].

Nicolas는 사용자의 RtlAllocateHeap() 함수 호출을 통해 새로이 생성되는 Heap이 아니라 프로세스에 존재하는 Default Heap을 공격하는 기술을 제안하였다.

아래 그림 71은 Default Heap의 구조를 나타낸다.

Chunk Header	
0	X
A	B
0	0
?	?

그림 71. Default Heap Chunk

Default Heap은 40byte의 크기를 가지고 있으며 그림 24의 A 필드는 다음 Default Heap을, B 필드는 이전 Default Heap를 가리킨다. X 필드는 Critical Section 구조체를 가리키는데 Critical Section 초기화 시에 Default Heap이 생성되어 X 필드가 초기화 된 Critical Section을 가리키게 된다. 여기서는 이것을 Linking Structure라고 부르며 이 Linking Structure와 Critical Section 사이의 관계는 아래 그림과 같다.

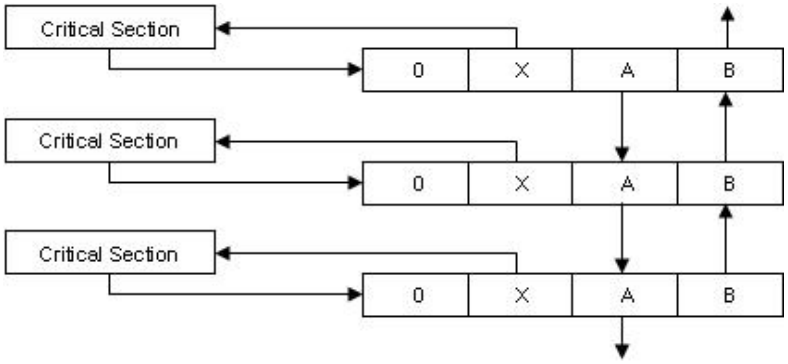

그림 72. Linking Structure

생성된 Critical Section이 파괴될 때 RtlDeleteCriticalSection()이 실행되는데 아래와 같은 코드를 포함하고 있다.

```
...
mov [eax], ecx          ; eax = B
mov [ecx + 4], eax      ; ecx = A
...
```

그림 73. RtlDeleteCriticalSection()

RtlDeleteCriticalSection()이 실행될 때 [3.4.8]에서 살펴 본 Window Heap 보호를 위한 어떤 검증도 하지 않기 때문에 [3.2.7]에서 살펴본 기술을 이용하여 임의의 코드를 실행할 수 있다.

3.5.6 Attack Fedora Core Linux

Windows 운영체제에 대한 해커들의 공격이 집중되던 때에 국내의 한 해커 Xpl017Elz는 2005년 후반기부터 최근까지 Fedora Core Linux에 대한 연구를 시작하여 각 버전에 대한 공격 방법을 차례로 발표한다[62]. 주로 RTL 공격을 이용해 각 버전 별로 다른 기능을 가지고 있는 Exec Shield 하에서 Overflow를 성공시킬 수 있는 방법을 다루고 있으며, 기존에 알려진 기술들을 조합하거나 재배치 조합과 같은 새로운 아이디어를 사용하여 Fedora Core Linux를 공략하고 있다. 재배치 조합을 이용한 Fedora Core Linux 8의 공격법은 [63]에 잘 나와있다. 이 외에도 Fedora Core Linux에서 Exec shield를 우회하는 기술을 다룬 [66], [67]과 같은 문서도 있지만 모두 [3.3.6]에서 이미 다루어진 기술들을 Fedora Core Linux에 적용시킨 것이다.

3.5.7 Attack Windows Vista

Microsoft가 2007년 1월에 출시한 Windows Vista는 보안에 중점을 두고 다시 만들어진 운영체제이며 NT 운영체제들과 완전히 다른 커널 구조를 가진다. Windows Vista는 정식

버전이 출시되기 전부터 많은 해커들의 공격 대상이 되었고 또 여러 취약점들이 발견되었다. 모든 바이너리에 대해 /GS 옵션으로 컴파일 했다는 Microsoft측의 주장과는 달리 많은 수의 바이너리 파일들이 /GS 옵션 없이 컴파일 되어 있음이 밝혀졌고[69], Vista에서부터 새롭게 적용된 ASLR이 Microsoft 밝힌 256가지 경우의 수를 가지는 것이 아니라 실제로는 32번 정도의 경우의 수를 가지는 것으로 밝혀졌다[70]. 또한 /GS 옵션이 몇몇 특정한 형태의 스택에 대해서는 적용되지 않음과 Windos XP에서도 발견되었던 PEB 랜덤화의 문제가 여전히 고쳐지지 않았음도 밝혀지게 된다[71]. 그 외에도 커널 보호를 위해 만들어진 PatchGuard, 드라이버의 무결성을 확인하기 위한 Driver Signing 등을 우회할 수 있는 기술들도 발표되었으나 본 문서의 범위를 벗어난다.

3.5.8 기타 공격 및 방어 기술

지금까지 살펴 본 버퍼 오버플로우 취약점에 대한 공격 및 방어 기술 외에도 SEH Overwrite를 막기 위한 기술[47], LibSafe를 우회하는 기술에 대한 연구[68], Integer Overflow 취약점을 자동으로 막는 기술에 대한 연구[51], [54], 함수를 호출하지 않고 임의의 코드를 실행시키는 기술[65], /GS 보호기술을 우회하는 새로운 기술에 대한 연구[67]등이 발표되었다.

5. 결론

지금까지 20여 년간 발전되어 온 공격 및 방어 기술들을 살펴보았다. 버퍼 오버플로우 취약점이 널리 알려진 지 20년이나 지난 지금에서도 여전히 버퍼 오버플로우 취약점은 발생하고 있으며 이를 방어하기 위한 기술 역시 계속 연구되고 있다. 이런 공격과 방어 기술은 대체로 아래의 특징을 가지며 발전하는 경향을 보이고 있다.

가. 기술의 난이도 및 복잡성이 점점 증가하는 경향을 보인다.

나. 공격 기술 및 방어 기술은 서로 먹이사슬 관계를 이루면서 발전한다..

다. 다양한 종류의 기술을 공격 또는 방어하기 위해 여러 기술들이 결합된 형태를 이루게 된다.

라. 방어 기술이 운영체제에 구현된 이후 리버스 엔지니어링 기술의 발전으로 운영체제의 내부 구조에 대한 연구가 이루어지면서 이를 이용한 공격 기술이 나타나게 된다.

또 한가지 재미있는 사항은 최근 운영체제에 기본으로 탑재되는 방어 기술인 Exec-Shield나 W^X, PaX에 구현된 기술들이 실제로는 이미 2000경에 이미 구현된 것들임을 볼 때 버퍼 오버플로우에 대한 방어 기술은 공격 기술에 비해 그다지 발전하지 않았음을 알 수 있다.

참 고 자 료

[1] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”, SANS 2000, Orlando FL, March 2000.

[2] Gary McGraw, John Viega, “Protect your code through defensive programming”. http://www.ibm.com/developerworks/library/s-buffer-defend.html

[3] Aleph1, “Smashing the stack (for Fun and Profit)”, *Phrack(49)*, November 1996.

[4] Solar Designer , Non-executable User Stack. http://www.openwall.com/linux/

[5] Solar Designer, “Getting around non-executable stack (and fix)”, August 1997, http://www.groar.org/expl/intermediate/ret-libc.txt

[6] Rafal Wojtczuk, “Defeating Solar Designer's Non-executable Stack Patch”, January 1998, http://insecure.org/sploits/non-executable.stack.problems.html

[7] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, “Stack Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”, Published in the proceedings of the 7th USENIX Security Symposium, January 1998, San Antonio, TX.

[8] Crispin Cowan, “ImmuniX OS Security Alert: Stack Guard 1.21 Released”, November 1999, http://seclists.org/bugtraq/1999/Nov/0119.html

[9] , matt Conover, “w00w00 on Heap Overflows”, w00w00 Security Team, http://www.w00w00.org/files/articles/heaptut.txt

[10] Todd C. Miller, Thed de Raadt, “strlcpy and strlcat - consistent, safe, string copy and concatenation”, In Proceedings of the 1999 USENIX Annual Technical Conference. http://www.usenix.org/event/usenix99/full_papers/millert/millert_html/index.html

[11] StackShield. http://www.angelfire.com/sk/stackshield

[12] klog, “The Frame Pointer overwrite”, *Phrack(55)*. http://www.phrack.org/issues.html?issue=55&id=8#article

[13] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, “A first step towards automated detection of buffer overrun vulnerabilities”, In Proceeding 7th Network and Distributed System Security Symposium.

[14] Bulba and Kil3r, “Bypassing Stack Guard and Stackshield”, *Phrack(56)*. http://www.phrack.org/issues.html?issue=56&id=5#article

[15] twitch, “Taking Advantage of non-terminated Adjacent Memory Spaces”,

Phrack(56).
http://www.phrack.org/issues.html?issue=56&id=14#article
[16] Arash Barstloo, Navjot Singh, and Timothy Tsai, “Transparent Run-Time Defense Against Stack Smashing Attacks”, Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference, 2000.
[17] Hiroaki Etoh and Kuuikazu Yoda, “Protecting from stack-smashing attacks”, IBM Research Division, Tokyo Research Laboratory, June 19, 2000.
http://www.trl.ibm.com/projects/security/ssp/main.html
[18] Securityfocus, “Wu-Ftpd Remote Format String Stack overwrite Vulnerability”, June 22, 2000.
http://www.securityfocus.com/bid/1387
[19] Pascal Bouchareine, “Format String vulnerability”, July, 2000.
http://doc.bughunter.net/format-string/format-buf.html
[20] grsecurity team , “Implementing non executable rw pages on the x86”.
http://pax.grsecurity.net/docs/pageexec.old.txt
[21] Juan Bello Rivas, “Overwriting the .dtors section”.
http://www.synnergy.net/downloads/papers/dtors.txt
[22] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman, “FormatGuard: Automatic Protection From printf Format String Vulnerabilities”, In Proceedings of the 2001 USENIX Security Symposium, August 2001, Washington DC.
http://crispincowan.com/~crispin/formatguard.pdf
[23] DilDog, cDc Ninja Strike Force, “The Tao of Windows Buffer Overflow”.
http://www.cultdeadcow.com/cDc_files/cDc-351/
[24] Solar Designer, “JPEG COM Marker Processing Vulnerability in Netscape Browsers”.
http://www.openwall.com/advisories/OW-002-netscape-jpeg/
[25] Peter Silberman and Richard Johnson, iDEFENSE, “A comparison of Buffer Overflow Prevention Implementations and Weaknesses”
http://www.phrack.org/issues.html?issue=56&id=8#article
[26] Michel MaXX, Kaemph, “Vudo: An Object Superstitiously Believed to Embody Magical Powers”, *Phrack(57)*
http://www.pharck.org/issues.html?issue=57&id=8#article
[27] anonymous, “Once upon a free()..”, *Phrack(57)*
http://www.pharck.org/issues.html?issue=57&id=9#article
[28] Lamagra, “Project OMEGA”, *corezine(2)*

http://www.ouah.org/omega11am.txt
[29] Lamagra, “The OMEGA project finished”, *corezine(3)*
http://www.ouah.org/omega.txt
[30] Richard Kettlewell , “Protecting Against Some Buffer-Overrun Attacks”.
http://www.greenend.org.uk/rjk/random-stack.html
[31] scut, Team Teso, “Exploiting Format String Vulnerabilities”
http://www.cs.utexas.edu/~shmat/courses/cs378_fall07/formatstrings.pdf
[32] Nergal, “The advanced return-into-lib(c) exploits: PaX case study”, *phrack(58)*
http://www.phrack.org/issues.html?issue=58&id=4#article
[33] Tyler Durden, “Bypassing PaX ASLR protection”, *pharck(59)*
http://phrack.org/issues.html?issue=59&id=9
[34] Umesh Shankar, Kunal Talwar, Jeffery S. Foster, David Wagner, “Detecting Format String Vulnerabilities with Type Qualifiers”, In 10th USENIX Security Symposium.
[35] John Viega, J.T. Bloch, Yoshi Kohno, Gary McGraw, “ITS4 : A Static Vulnerability Scanner for C and C++ Code”, Annual Computer Security Applications Conference. December 2000.
[36] RATS, http://www.fortify.com/security-resources/rats.jsp
[37] blexim, “Basic Integer Overflows”, *Phrack(60)*.
http://www.phrack.org/issues.html?issue=60&id=10
[38] T Jim, G Morrisett, D Grossman, M Hicks, J Cheney, Y Wang, “Cyclone: A safe dialect of C”, In USENIX Annual Technical Conference
[39] G C Necula, S McPeak, W Weimer, “CCured: Typesafe retrofitting of legacy code”, In Proceedings of the Principles of Programming Languages. ACM
[40] David Litchfield, Director of Security Architecture @stack, “Windows 2000 Format String Vulnerabilities”.
http://www.ngssoftware.com/papers/win32format.doc
[41] Pax Project. http://pax.grsecurity.net/docs/
[42] Announce Exec Shield.
http://www.redhat.com/f/pdf/rfel/WHP0006US_Execshield.pdf
[43] beist, “Fedora Core2에서 Exec-Shield를 우회하여 Stack 기반 Overflow 공격기술 한번에 성공하기”.
http://beist.org/research/public/fedora/index.html
[44] Xpl017Elz , “glibc-2.5/elf/dl-runtime.c 코드 분석을 통한 lazy binding 재배치 방식 분석”.
http://x82.inetcop.org/h0me/papers/FC_exploit/relocation.htm

[45] Jerald Lee, “SEH Overwrite”.
http://lucid7.egloos.com/1436052
[46] David LitchField, “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server”, SEPT. 2003,
http://www.ngssoftware.com/papers/defeating-w2k3-protection.pdf
[47] skape, “Preventing the exploitation of seh overwrites”, *Uniformed(5),* SEPT, 2006.
http://www.uninformed.org/?v=5&a=2&t=sumry
[48] Windows XP Service Pack 2 Overview White Paper. Microsoft
http://download.microsoft.com/download/6/6/c/66c20c86-dcbe-4dde-bbf2-ab1fe9130a97/windows xp sp 2 white paper.doc
[49] Steve Friedl, “Analysis of Microsoft XP Service Pack 2”.
http://www.unixwiz.net/techtips/xp-sp2.html
[50] Halvar Flake, “Third Generation Exploitation”, BlackHat USA 2002
[51] David Brumley, Dawn Song, Joseph Slember, “Towards Automatically Eliminating Integer-Based Vulnerabilities”, Carnegie Mellon University, Tech. Rep. CMU-CS-06-136
http://undocumented.ntinternals.net/
[52] David Litchfield , “Winows Heap Overflow”, BlackHat USA 2004,
http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt
[53] eEye Digital Security, “Microsoft Internet Information Services Remote Buffer Overflow”.
http://research.eeye.com/html/advisories/published/AD20010618.html
[54] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, Dawn Song, “RICH: Automatically protecting against integer-based vulnerabilities”, In Symp. On Network and Distributed Systems Security
[55] Puttaraksa , “Heap Spray”.
http://sf-freedom.blogspot.com/2006/06/heap-spraying-introduction.html
[56] Alexander Anisimov, Positive Technologies, “Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass”.
http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf
[57] skepe, Skywing, “Bypassing Windows Hardware-enforced Data Execution Prevention”, *Uniformed(2),* SEPT, 2005.
http://www.uninformed.org/?v=2&a=4&t=sumry
[58] “W^X”, OpenBSD

http://www.openbsd.org/papers/ven05-deraadt/mgp00009.html
[59] cyberterror, “A DEP evasion technique”.
http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html
[60] David Litchfield, “Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms(XPMs) on the Windows platform”.
http://www.ngssoftware.com/papers/xpms.pdf
[61] Nicolas Falliere, “Bypassing Windows heap protections”.
http://packetstormsecurity.nl/papers/bypass/bypassing-win-heap-protections.pdf
[62] Xpl017Elz, “Fedora Core, CentOS, Whitebox Linux (Exec-shield) exploit documents”.
http://x82.inetcop.org/h0me/papers/FC_exploit/
[63] Xpl017Elz, “Apache Tomcat Connector jk2-2.0.2 (mod_jk2) Remote Overflow Exploit”.
http://www.milw0rm.com/exploits/5386
[64] Executable and Linkable Format (ELF), Portable Formats Specification, Version 1.1
http://pds8.egloos.com/pds/200803/30/74/elf_format.pdf
[65] Hovav Shacham, “The Geometry of innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”, ACM Conference on Computer and Communications Security 2007: 552-561
[66] Oded Horovitz, “Big Loop Integer Protection”, *phrack(60)*
http://www.phrack.org/issues.html?issue=60&id=9
[67] skape, “reducing the effective entropy of gs cookies”, *Uniformed(7),*
MAY, 2007.
[68] Overflow.pl, “Libsafe – Safety Check Bypass Vulnerability”, MAY, 2005
http://www.overflow.pl/adv/libsafebypass.txt
[69] Ollie Whitehouse, Symantec Advanced Threat Research, “Analysis of GS protections in Microsoft Windows Vista”,
www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf
[70] Ali Rahbar, “An analysis of Microsoft Windows Vista’s ASLR”
www.sysdream.com/articles/Analysis-of-Microsoft-Windows-Vista's-ASLR.pdf
[71] Ollie Whitehouse, Symantec Advanced Threat Research, “GS and ASLR in Windows Vista”
www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf

부 록

아래는 본 문서를 작성하기 위해 조사했던 각 기술들을 연도별로 정리한 것이다. 보기 좋게 표로 정리하고 싶었으나 귀차니즘의 압박으로 그냥 붙여넣기로 끝내버렸다.

1992년: Purify: Fast Detection of Memory Leaks and Access Errors. R.Hastings and B.Joyce. In Proceedings of the Winter USENIX Conference.

1995년: 7월, Bounds Checking for C, R.Jones and P.Kelly
http://www-ala.doc.ic.ac.uk/phjk/BoundsCheckin.html

1995년: 10월, L0pht 의 Mudge, How to wirte Buffer Overflows 발표
http://insecure.org/stf/mudge_buffer_overflow_tutorial.html

1996년: 11월, Smashing the stack(for Fun and Profit), Aleph1, Phrack 49

1996년: A secure environment for untrusted helper applications. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. In Proceedings of the 6th USENIX Security Symposium

1997년: FreeBSD stack integrity patch, A.Snarskii
ftp://ftp.lucky.net/pub/unix/local/libc-letter

1997년: 8월, Non-Executable 스택발표(개선), Getting around non-executable stack (and fix) Solar Designer.
http://www.groar.org/expl/intermediate/ret-libc.txt

1998년: 1월(January), StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang.
http://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

1998년: "Protecting Systems from Stack Smashing Attacks with StackGuard", In Proceedings of the 5th Linux Expo

1998년: 1월, Defeating Solar Designer's Non-executable Stack Patch.

http://insecure.org/sploits/non-executable.stack.problems.html

1998년: 4월, The Tao of Windows Buffer Overflow.
http://www.cultdeadcow.com/cDc_files/cDc-351/

1998년: 5월 May, 스택가드 개선, StackGuard 1.1: Stack Smashing Protection for Shared Libraries. In IEEE Symposium on Security and Privacy, Oakland, CA, May 1998. Brief presentation and poster session.

1998년: 8월, Protection Against Some Buffer-Overrun Attacks, Richard kettlewell.
http://www.greenend.org.uk/rjk/random-stack.html

1999년: 1월, w00woo on Heap Overflows.
http://www.w00w00.org/files/articles/heaptut.txt

1999년: 6월, strlcpy and strlcat — consistent, safe, string copy and concatenation.
http://www.usenix.org/event/usenix99/full_papers/millert/millert_html/index.html

1999년: 8월, Stack Shield 릴리즈
Stack shield: A "stack smashing" technique protection tool for linux, "Vendicator"
http://www.angelfire.com/sk/stackshield/

1999년: 8월, Project OMEGA, Lamagra, corezine #2
http://www.ouah.org/omega11am.txt

1999년: 9월, klog, "The Frame Pointer overwrite", Phrack Magazine 55 article 8.
http://www.phrack.org/issues.html?issue=55&id=8#article

1999년: 9월, dark spyrit "Win32 Buffer Overflows".
http://www.phrack.org/phrack/55/p55-15

1999년: 10월, "The OMEGA project finished", Lamagra, corezine #3
http://www.ouah.org/omega.txt

2000년: 2월, A first step towards automated detection of buffer overrun vulnerabilities. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken.

In Proceeding 7th Network and Distributed System Security Symposium

2000년: 5월, Bypassing Stackguard and Stackshield, Bulba and Kil3r, Phrack 56.
http://www.phrack.org/issues.html?issue=56&id=5#article

2000년: 5월, TAKING ADVANTAGE OF NON-TERMINATED ADJACENT MEMORY SPACES(strncpy), phrack56, twitch.
http://www.phrack.org/issues.html?issue=56&id=14#article

2000년: 5월, rix, SMASHING C++ VPTRS, phrack56, rix.
http://www.phrack.org/issues.html?issue=56&id=8#article

2000년: 6월, Transparent Run-Time Defense Against Stack Smashing Attacks, A. Baratloo, N.Singh, and T.Tsai. In Proceedings of the USENIX Annual Technical Conference LIBSAFE
http://www.usenix.org/event/usenix2000/general/full_papers/baratloo/baratloo.pdf

2000년: 6월, IBM의 Hiroaki Etoh가 GCC의 확장판으로써 Stackguard를 변형시킨 propolice 를 설계함. GCC extension for protection applications from stack-smashing attacks.
http://www.trl.ibm.com/projects/security/ssp/main.html

2000년: 7월, Format string vulnerability, Pascal Bouchareine,
http://doc.bughunter.net/format-string/foramt-buf.html

2000년: 7월 25일, JPEG COM Marker Processing Vulnerability in Netscape Browsers, Solar Designer.
http://www.openwall.com/advisories/OW-002-netscape-jpeg/

2000년: 9월, PaX "Implementing non executable rw pages on the x86"

2000년: 12월, John Viega, J.T. Bloch, Tadayoshi Kohno and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. Annual Computer Security Applications Conference. December 2000.

2001년: David Litchfield, Windows 2000 Format String Vulnerabilities.

http://community.corest.com/~juliano/win32format.doc

2001년: 3월, Juan Bello Rivas, "Overwriting the .dtors section".
http://www.synnergy.net/papers/dtors.txt

2001년: 5월, Cqual, "Detecting Format String Vulnerabilities with Type Qualifiers", Umesh Shankar, Kunal Talwar, Jeffery S. Foster, David Wagner, In 10th USENIX Security Symposium, August 2001

2001년: 8월, crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In Proceedings of the 2001 USENIX Security Symposium, August 2001, Washington DC.
http://crispincowan.com/~crispin/formatguard.pdf

2001년: 8월, "Statically Detecting Likely Buffer Overflow Vulnerabilities", David Larochelle and David Evans. In 2001 USENIX Security Symposium, Washington, D.C., August 13-17, 2001

2001년: 8월, Michel Kaempf, Vudo malloc tricks.
http://www.phrack.org/phrack/57/p57-0x0b

2001년: 8월, Once upon a free() 프랙57, anynymous.
http://www.phrack.org/phrack/57/p57-0x0c

2001년: 9월, scut, Team Teso, "Exploiting Format String Vulnerabilities"

2001년: 12월, The advanced return-into-libc exploits: PaX case study, phrack 58

2002년: 7월, Bypassing PaX ASLR protection, phrack59.
http://www.phrack.org/phrack/59/p59-0x09

2002년: 7월, riq and gera, Advances in format string exploitation, phrack59

2002년: 7월, "Third Generation Exploitation : Smashing the Heap under Win2k", Halvar Flake, BlackHat USA 2002

2002년: 12월, blexim "Basic Integer Overflows".
http://www.phrack.org/phrack/60/p60-0x0a.txt

2002년: 12월, "Big Loop Integer Protection", Oded Horovitz, phrack 60

2002년: 12월 "Smashing the Kernel Stack for Fun and Profit", phrack 60

2003년: 5월, Exec-Shield, Redhat.

2003년: 9월, Litchfield, David. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server

2004년: 7월, "Winows Heap Overflow", David Litchfield, BlackHat USA 2004.
http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

2004년: 8월, Windows XP Service Pack2.

2004년: 10월, "Fedora core2에서 Exec-Shield를 우회하여 Stack 기반 Overflow 공격 기법 한번에 성공하기", beist,
http://beist.org/research/public/fedora/index.html

2004년: 11월, "Heap Spray",
http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php

2005년: 1월 Anisimov, Alexander. Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass

2005년: 4월, Libsafe - Safety Check Bypass Vulnerability.
http://www.overflow.pl/adv/libsafebypass.txt

2005년: 7월, "Bypassing Windows Hardware-enforced Data Execution Prevention", skape, Skywing.
http://www.uninformed.org/?v=2&a=4&t=sumry

2005년: 9월, "Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms(XPMs) on the Windows platform", David Litchfield.
http://www.ngssoftware.com/papers/xpms.pdf

2005년: 11월, "Bypassing Windows heap protections", Nicolas Falliere.
http://packetstormsecurity.nl/papers/bypass/bypassing-win-heap-protections.pdf

2006년: 1월, "bypassing patchguard on windows x64", skape, Skywing
http://www.uninformed.org/?v=3&a=3&t=sumry

2006년: 7월, "preventing the exploitation of seh overwrites", skape,
http://www.uninformed.org/?v=5&a=2&t=sumry

2006년: March, "Towards Automatically Eliminating Integer-Based Vulnerabilities", David Brumley, Dawn Song, Joseph Slember. Carnegie Mellon University, Tech. Rep. CMU-CS-06-136

2007년: 1월 Windows Vista 출시

2007년: "RICH: Automatically protecting against integer-based vulnerabilities", David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, Dawn Song, In Symp. On Network and Distributed Systems Security

2007년: 1월, "subverting patchguard version 2", Skywing
http://www.uninformed.org/?v=6&a=1&t=sumry

2007년: 5월, "reducing the effective entropy of gs cookies", skape
http://www.uninformed.org/?v=7&a=1&t=sumry

2007년: 7월, "The Geometry of innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", Hovav Shacham

2008년: 9월, "patchguard reloaded: a brief analysis of patchguard version 3", Skywing
http://www.uninformed.org/?v=8&a=5&t=sumry