|  |  |  |  |
| --- | --- | --- | --- |
| **과제명** | 포맷스트링 & 버퍼오버플로우 | | |
| **팀명** | 8조 | **작성일자** | 2017.11.26 |
| **팀원** | 배동현 (20123214) | | |
| 김민지 (20176063) | | |
| 정유정 (20176064) | | |

1. 연구방법

코드보안의 종류인 포맷스트링(Format String) 공격과 버퍼오버플로우(Buffer Overflow) 공격에 대해 알아보고 보안법에 대해서도 알아본다.

1. 연구과정
2. 포맷스트링(Format String)

* 포맷 스트링(Format String)이란 일반적으로 사용자로부터 입력을 받아들이거나 결과를 출력하기 위하여 사용하는 형식이다. 예를 들어, printf() 함수에서 숫자, 문자 등을 출력할 때 %d, %f, %c, %x, %p, %n과 같은 포맷 스트링이 사용된다. 즉, printf(“%d”, i)와 같이 사용하면 변수 i의 값을 출력할 수 있다.

하지만, printf(“%x”)처럼 변수없이 쓰게 되면 메모리의 값들을 순서대로 출력하게 되어 메모리 구조를 파악하는데 악용될 수 있다. %n이나 %hn를 이용하면 특정 메모리 위치의 값을 다른 것으로 변경할 수 도 있다. 이렇게 포맷 스트링 사용시의 오류를 이용하여 시스템의 권한을 획득하는 기법을 포맷 스트링 공격이라 한다. 포맷 스트링 공격 printf() 등의 C함수뿐 아니라 함수간에 값을 전달하는 선언이 경우이다.

* 일반적인 익스플로잇은 이러한 기법들의 조합을 사용해서 프로그램이 라이브러리 함수의 주소나 스택의 반환 주소에 악의적인 쉘코드를 가리키는 포인터를 겹쳐쓰게 한다. 포맷 명시자들에 대한 패딩 파라미터들은 많은 바이트들의 출력을 제어하는데 사용되며 %x 토큰은 자신이 도달하는 포맷 스트링의 시작까지 스택으로부터 바이트들을 pop하는데 사용된다. 포맷 스트링의 시작은, %n 포맷 토큰이 악의적인 코드의 주소로 겹쳐쓸 수 있는 주소를 포함할 수 있게 다듬어 진다.

포맷 버그들이 이전에 위험하지 않게 여겨졌고 많은 일반적인 툴들의 결과로 나왔기 때문에 이것은 흔한 취약점이다. MITRE의 CVE 프로젝트는 2007년 6월에 500개의 취약한 프로그램들을 보여주었는데, 2001년과 2006년 사이에 9번째로 자주 보고된 취약점 타입이었다.

포맷 스트링 버그들은 프로그래머가 사용자가 제공한 데이터를 포함한 문자열을 print하고자 할 때 가장 흔하게 볼 수 있다. 프로그래머는 실수로 printf(buffer) 를 printf(“%s”, buffer) 대신에 쓸 수 있다. 첫 번째는 buffer를 포맷 스트링으로 해석하고 포함하는 어느 포매팅 명령어도 파스(parse)한다. 두 번째는 프로그래머의 의도와 같이 간단하게 문자열을 화면에 보여준다.

포맷 버그들은 C의 인자 전달 컨벤션들이 타입 안전하지 않아서 발생한다. 특히 varargs 메커니즘은 함수들이 어느 개수의 인자들이라도 받아들일 수 있게 하는데 이것은 원하는 만큼 콜 스택에서 인자들을 pop할 수 있기 때문이다.

포맷 스트링 버그들은 C외에도 다른 프로그래밍 언어들에도 발생할 수 있다. 하지만 훨씬 적게 발견되며 공격자가 원하는 코드를 실행할 수 있을 만큼 익스플로잇 되기도 힘들다.

1. 포맷인자



컴퓨터에서 어떤 크기의 데이터를 메모리에 저장할 때 바이트 단위로 나누어 저장한다.

1. 리틀 엔디언 & 빅 엔디언

: 엔디언(Endianness)은 컴퓨터의 메모리와 같은 1차원의 공간에 여러 개의 연속된 대상을 배열하는 방법을 뜻하며, 바이트를 배열하는 방법을 특히 바이트 순서(Byte order)라 한다.

엔디언은 보통 큰 단위가 앞에 나오는 빅 엔디언(Big-endian)과 작은 단위가 앞에 나오는 리틀 엔디언(Little-endian)으로 나눌 수 있으며, 두 경우에 속하지 않거나 둘을 모두 지원하는 것을 미들 엔디언(Middle-endian)이라 부르기도 한다.

빅 엔디안은 최상위 바이트(MSB – Most Signficant Byte) 부터 차례로 저장하는 방식이며, 리틀 엔디안은 최하위 바이트(LSB – Least Significant Byte) 부터 차례로 저장하는 방식이다.

CPU 아키텍처에 따라 바이트 저장순서가 달라질 수 있기 때문에 두 가지로 나누어지는데 그것이 바로 리틀-엔디안 과 빅-엔디안 방식이다. 어떤 CPU에서는 이 두 가지 방식을 모두 지원하도록 구성할 수도 있다.





“ “ 안에 포함되어 있는 “Hello, %s\n”이 Format String이다.

즉 Format String은 이 Format String을 사용하는 함수에 대해, 어떤 형식 혹은 형태를 지정해 주는 문자열을 의미한다.

1. Format String 사용시 문제점

: 포맷스트링 공격 역시 대부분의 다른 취약점 버그 또는 버그들처럼 일반 프로그래머들의 작은 실수에서 발생된 취약점을 이용하는 것이다.

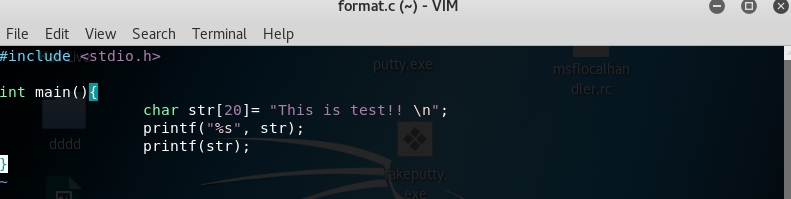


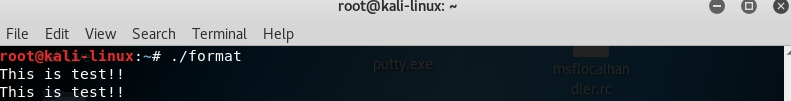
하지만, 어떤 프로그래머들은 위와 같은 형태를 이용하지 않고 프로그래밍을 보다 편하게 하기 위해서 다음과 같이 사용하는 경우가 있다.



이와 같은 형태의 프로그래밍이 잘못된 것은 아니다. 어떻게 보면 같은 기능을 수행하는 두가지의 코드 중 2번의 소스코드가 1번의 소스코드보다 적은 양의 소스코드를 사용한다.

따라서 2번의 소스코드형태가 좀 더 현명한 소스코드로 보일 수 있다. 그러나 2번의 소스코드를 이용하여 프로그래밍하는 경우에는 해커들에게 프로그램의 흐름을 바꿀 수 있는 기회를 제공하게 된다.



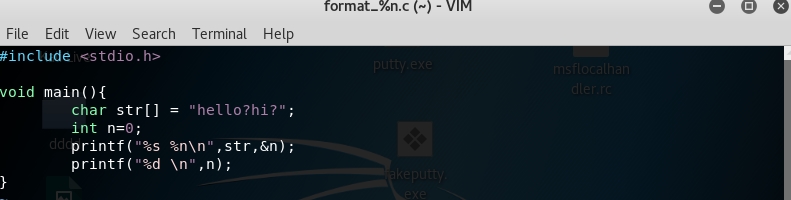


위와 같이 문제 없이 같은 기능을 수행하는 것을 확인할 수 있다.

그러나 printf 함수에 의해서 해석되는 문자열 “str”은 출력하고자 하는 문자열이 아니라 printf 함수에서 사용할 각종 형식 지시자(%d, %s, %c 등)를 포함한 Format String 으로 인식하게 된다.

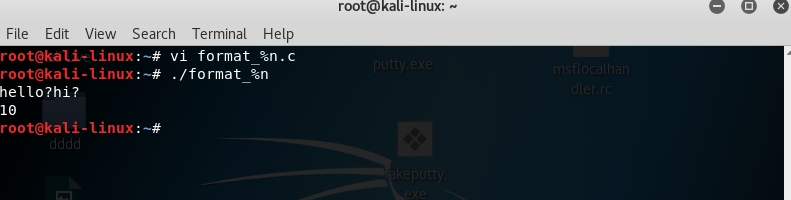
1. 새로운 지시자(directive) %n

Format String에 사용되는 형식 지시자 들 중에는 출력 될 문자들의 개수를 셀 수 있는 기능을 제공하는 것이 %n이라는 형식 지시자 이다. %n이라는 형식 지시자를 사용하면, %n이 사용되기 직전에 사용된 형식에 의해 출력된 문자들의 개수가 다음 변수에 저장된다.



%n 이전까지의 문자들의 수를 세어서 &n으로 저장한다,

%s는 “hello?hi?” 9자리 와 “ “ 공백 1자리로 (텍스트 9자리) + (공백1자리) 총 10이 n에 저장된다. 즉, %n 형식은 Format String내에서 %n 지시자 전에 지정된 출력 되어져야 하는 모든 공간의 개수를 해당 변수로 저장하게 된다.



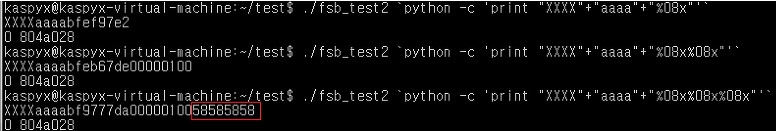
1. Format String 버그 취약점 접근하기

Format String 버그 취약점 공략을 쉽게 접근하기 위해 간단한 코드를 추가해준다.



문자를 입력받아 배열 buf에 저장하여 값 출력과 %n을 이용하여 전역변수 p의 값을 변경하기 위해 만든 코드이다. 컴파일 옵션과 고정 스택 옵션을 사용하였다면 크게 메모리 값이 크게 달라지는 점은 없었을 것이다.

Format String을 입력하여 내가 어떤 주소를 수정할지, printf 함수 기준으로 스택에 쌓인 파라미터를 추출하여 계산을 해보았다.



1. Python Script문을 이용하여 “XXXX”(아스키코드 값으로 0x58)가 몇 번째에 있는지 확인하는 방법이다.
2. 16 진수로 출력할 때 8자리로 맞추기 위하여 %08x로 사용하였다.
3. %08x \*3 하였을 때 XXXX가 출력되는 걸 확인할 수 있다.

XXXX (58585858) 라는 문자열은 세 번째 %08x를 넣었을 때 출력되었다. (%08x는 16진수로 출력할 때 8자리가 안되어도 자릿수를 맞추기 위해서 지정해준다.)

XXXX라는 값을 수정하고 싶으면 세 번째 %08x를 %n으로 바꿔주면 된다.



1. XXXX라는 문자열 대신 우리가 쓸 수 있는 메모리 주소를 지정하고, 문자열의 수를 조절하면 원하는 메모리주소에 원하는 값을 쓸 수 있다.

CPU가 intel사이기 때문에 메모리 주소를 거꾸로 사용하는 리틀 엔디안 방식을 사용한다.

1. 0804a028 ---🡪 28a00408
2. \x28\xa0\x04\08 (4자리) + “aaaa” (4자리) + %08x%08x (16자리) = 24 🡪 16진수로 0x18 이므로 p값은 0에서 18로 변경된다.
3. 덮어쓸 메모리 주소를 정확하게 알고 있다면 원하는 값을 해당 메모리 주소에 덮어쓸 수 있는 것이다. 응용을 해서 한다면 메모리를 파괴하거나 악성코드의 주소를 return주소에 덮어서 공격자가 원하는 행위를 하게 만들 수도 있다.

포맷스트링 공격 보안법

1.printf문 정상적으로 사용하기

printf("%s \n", buffer);

2.스트링 공격에 취약한 함수를 사용하지 않기

1)int fprintf(fp, const char \*fmt,..)

2)int sprintf(char \*str, const char \*fmt, ..)

3)int snprintf(char \*str, size\_t count, const char \*fmt, ..)

3.시스템 패치를 꾸준히 해준다

-> 커널이 발전하면서 SETUID에 대한 보안설정 사항이 늘어난다

버퍼 오버플로우(Buffer Overflow)

* 버퍼 오버플로우 또는 버퍼 오버런(Buffer OverRun)은 메모리를 다루는 데에 오류가 발생하여 잘못된 동작을 하는 프로그램 취약점이다. 컴퓨터 보안과 프로그래밍에서 이는 프로세스가 데이터를 버퍼에 저장할 때 프로그래머가 지정한 곳 바깥에 저장하는 것이다. 벗어난 데이터는 인접 메모리를 덮어 쓰게 되는데 다른 데이터가 포함되어 있을 수도 있는데, 손상을 받을 수 있는 데이터는 프로그램 변수와 프로그램 흐름 제어 데이터도 포함된다. 이로 인해 잘못된 프로그램 거동이 나타날 수 있으며, 메모리 접근 오류, 잘못된 결과, 프로그램 종료, 또는 시스템 보안 누설이 발생할 수 있다.

버퍼 오버플로우가 코드를 실행시키도록 설계되거나 프로그램 작동을 변경시키도록 설계된 입력에 의해 촉발될 수 있다. 따라서 이는 많은 소프트웨어 취약점의 근간이 되며 악의적으로 이용될 수 있다. 경계 검사로 버퍼 오버플로우를 방지할 수 있다.

버퍼 오버플로우는 보통 데이터를 저장하는 과정에서 그 데이터를 저장할 메모리 위치가 유효한지를 검사하지 않아 발생한다. 이러한 경우 데이터가 담긴 위치 근처에 있는 값이 손상되고 그 손상이 프로그램 실행에 영향을 미칠 수도 있다. 특히, 악의적인 공격으로 인해 프로그램에 취약점이 발생할 수도 있다.

흔히 버퍼 오버플로우와 관련되는 프로그래밍 언어는 C와 C++로 어떤 영역의 메모리에서도 내장된 데이터 접근 또는 덮어쓰기 보호 기능을 제공하지 않으며 어떤 배열에 기록되는 데이터가 그 배열의 범위 안에 포함되는지 자동으로 검사하지 않는다.

* 버퍼 오버플로우는 불충분한 바운드 확인에 의해 버퍼에 쓰인 데이터가 버퍼에 이미 할당된 근접한 메모리 주소에 있는 데이터 값을 오염시킬 때 발생한다. 대부분 이는 문자열을 하나의 버퍼에서 다른 버퍼로 복사할 때 발생한다.
* 보호 대응 수단

: 다양한 기법이 버퍼 오버플로우를 감지 또는 방지하기 위하여, 다양한 대가를 지불하며 사용되어 왔다. 가장 신뢰성 있는 버퍼 오버플로우를 회피 또는 방지하기 위한 방법은 언어 수준에서 자동 보호를 사용하는 것이다. 그러나 이러한 종류의 보호는 기존 코드에 적용할 수 없고, 때때로 기술적, 상업적, 또는 문화적 제한으로 취약한 언어를 요구한다

1. 프로그래밍 언어 선택

: 프로그래밍 언어 선택은 버퍼 오버플로우 발생에 깊이 있는 영향을 미칠 수 있다.

1. 안전한 라이브러리 사용

: 버퍼 오버플로우의 문제는 C와 C++언어에서는 일반적이다. 왜냐하면 이들 언어가 데이터 형을 보관 장소로서 버퍼의 저수준 표현상 상세를 노출시키기 때문이다. 버퍼 오버플로우는 반드시 따라서 버퍼 관리 코드 안에서의 높은 수준의 정확성을 유지함으로써 회피되어야 한다. 오래 전부터 표준 라이브러리 함수 중 경계를 검사하지 않는 함수의 사용을 피할 것이 권고되어 왔다: gets, scanf, strcpy등. 모리스 웜은 핑거 데몬 안에서 호출되는 gets 함수를 이용하였다.

1. 버퍼 오버플로우 보호

: 버퍼 오버플로우 보호는 가장 일반적인 버퍼 오버플로우를 검출하기 위해 사용되며 함수가 귀환할 때 스택이 변경되었는지 검사한다. 변경되었다면 프로그램이 세그멘테이션 오류를 발생 시키며 중단된다. 그러한 시스템의 예가 gcc 패치인 립세이프 LibSafe, 스택 가드, 프로폴리스 이다. 마이크로소프트의 데이터 실행 방지 모드는 명백히 SHE 예외 처리기를 가리키는 포인터를 덮어쓰기로 부터 보호한다.

1. 포인터 보호

: 버퍼 오버플로우는 포인터를 조작함으로써 작동한다. 포인트 가드는 컴파일러 확장으로 제안되었는데, 공격자가 신뢰성 있게 포인터와 주소를 조작하는 것을 방지한다.

1. 실행 공간 보호

: 실행 공간 보호는 버퍼 오버플로우를 방지하기 위한 접근 방법으로 스택이나 힙 상의 코드가 실행되는 것을 막는다. 공격자는 버퍼 오버플로우를 이용하여 임의의 코드를 프로그램의 메모리에 삽입할 수 있지만, 실행 영역 보호가 있다면, 그 코드를 실행하고자 하는 어떠한 시도도 예외를 발생시킬 것이다.

1. 주소 공간 배치 난수화

: 주소 공간 배치 난수화는 컴퓨터 보안 기능으로 중요 데이터 영역, 예를 들어 실행 코드의 기반 주소, 라이브러리, 힙, 스택 주소 등을 임의로 프로세서의 주소 공간에 배치하는 것이다. 함수와 변수를 찾을 수 있는 가상 메모리 주소의 난수화로 버퍼 오버플로 이용이 더 어려워지지만 불가능한 것은 아니다. 공격자가 이용 시도를 각각의 시스템에 따라 다르게 하도록 강요하므로 인터넷 웜 방어에 더 유용하다. 비슷하지만 덜 효과적인 방법은 프로세스와 라이브러리를 가상 주소 공간에 리베이스 하는 것이다.

1. 심층 패킷 조사

: 심층 패킷 조사로 네트워크 경계에서 아주 간단한 버퍼 오버플로우 원격 시도를 공격 고유 신호과 겸험적 방법으로 검출할 수 있다. 알려진 공격 고유 신호 또는 긴 NOP 명령이 검출되면 패킷을 막을 수 있으며, 공격 패킷 내용이 조금 달라도 사용할 수 있다.

1. 연구 결과

: 많은 컴파일러들이 정적으로 포맷 스트링을 검사하고 의심스러운 포맷에 관해서는 경고를 보여준다. GNU 컴파일러 모음에서 관련있는 컴파일러 플래그들로는, -Wall, -Wformat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral, 그리고 –Wformat=2가 있다.

이것들 대부분은 단지 컴파일 타임 시에 알려진 안좋은 포맷 스트링을 탐지 하는데에만 유용하다. 만약 포맷 스트링이 사용자나 애플리케이션의 외부소스에서 온다면, 애플리케이션은 반드시 그것을 사용하기 전에 검증을 하여야 한다. 또한 애플리케이션이 포맷 스트링을 그때 그때 생성하거나 선택한다면 주의가 필요하다. –Wformat-nonliteral 검사는 더 강력하다.

1. 참고자료

* 위키백과, ‘엔디언’,

https://ko.wikipedia.org/wiki/%EC%97%94%EB%94%94%EC%96%B8

* Tistory, ‘Format String 공격’,

<http://shayete.tistory.com/entry/5-Format-String-Attack-FSB>

* 정보전산원, ‘포맷 스트링 공격이란 무엇인가요’, <http://cc.wku.ac.kr/?p=489>
* Tistory, ‘포맷 스트링 공격’, <http://geundi.tistory.com/124>
* Tistory, ‘리틀 엔디언 & 빅 엔디언’, <http://genesis8.tistory.com/37>
* ‘포맷 스트링 공격’,

<http://www.hackerschool.org/HS_Boards/data/Lib_system/The_Mystery_of_Format_String_Exploitation.pdf>