**24.01.16 보고서**

**서강대학교 20191244 김현승**

* **Instrumenting programs**

AFL은 gcc나 clang을 대체하는 전용 컴파일러를 사용해 소스 코드 수준의 static instrumentation을 지원한다. 이 전용 컴파일러를 이용해 타겟 프로그램을 컴파일 하는 방법은 빌드 과정에 따라 다양하지만 일반적으로는 다음과 같다.

$ CC=/path/to/afl/afl-gcc ./configure

$ CXX=/path/to/afl/afl-g++ ./configure

$ make clean all

위와 같이 C 프로그램의 경우 CC, C++ 프로그램의 경우 CXX 환경 변수 값을 AFL의 전용 컴파일러 경로로 지정해 빌드하면 컴파일 시 instumentation 코드가 삽입된다.

위는 소스 코드가 제공된 경우 사용할 수 있는 instrumentation이다. 소스 코드가 제공되지 않았을 때 바이너리 파일만으로도 instrumentation을 적용할 수 있다. AFL은 QEMU 모드를 통해 이 기능을 지원하는데, AFL을 QEMU 모드로 설정하면 프로그램을 실행하는 동안 QEMU 에뮬레이터가 런타임에서 코드를 instrumenting하게 된다. 사용 방법은 다음과 같다.

$ cd /path/to/afl/qemu\_mode

$ ./build\_qemu\_support.sh

AFL의 qemu\_mode 디렉토리로 이동해 build\_qemu\_support.sh를 실행하면 자동으로 QEMU 모드가 빌드되어 QEMU 에뮬레이터를 이용한 instrumentation을 사용할 수 있게 된다. 이 모드는 컴파일 타임 instrumentation보다 약 2~5배 정도 느리고 취약점을 찾아내는데 한계가 있을 수 있다.

* **Fuzz Dact with AFL**

Dact는 입력 파일의 각각 블록을 분석하여 각 블록에 대해 최적의 알고리즘을 선택해 압축하는 프로그램이다. Dact는 0.8.42 버전에 취약점이 존재한다고 밝혀져 있다. 먼저 해당 버전의 Dact를 빌드해보자.

$ wget https://fossies.org/linux/privat/old/dact-0.8.42.tar.gz

$ tar -xvf dact-0.8.42.tar.gz

$ cd dact-0.8.42

위와 같이 dact-0.8.42.tar.gz 파일을 압축 해제 후 해당 폴더로 이동한다.

$ CC=/path/to/afl/afl-gcc ./configure

$ make

위에서 설명했듯이 CC 환경 변수 값을 afl-gcc의 경로로 변경하고 configure 스크립트를 실행함으로써 dact를 빌드할 때 instrumentation을 위한 코드가 삽입되도록 한다. 이후 make 명령어로 빌드하면 AFL을 이용할 수 있도록 dact가 빌드된다.

dact가 빌드되었으니 afl-fuzz를 이용해 실제로 퍼징을 진행해보자. afl-fuzz의 사용법은 다음과 같다.

$ ./afl-fuzz -i testcase\_dir -o findings\_dir /path/to/program [...params...]

testcase가 들어갈 input 디렉터리와 출력이 저장될 output 디렉터리가 필요하다. input directory에는 초기 testcase가 필요하므로 다음과 같이 testcase를 생성해본다.

$ mkdir input

$ echo hello > hello.txt

$ ./dact hello.txt

$ mv hello.txt.dct input/

이후 dact에 -dcf 옵션을 주고 fuzzing을 실행한다. dact의 -d 옵션은 압축 해제, -c 옵션은 압축(해제) 결과 출력, -f 옵션은 강제 압축(해제)이다. 이 프로그램의 취약점은 압축 해제 시 존재하므로 해당 옵션을 주는 것이다.

$ afl-fuzz -i input -o output -- ./dact -dcf

위 명령어를 실행하면,

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

위와 같이 fuzzing이 진행되고 현재 19800개의 crash, 38개의 unique crash가 발견됨을 확인할 수 있다(51분간 fuzzing 진행). fuzzing 중단 후 output directory로 이동해보면 queue, crashes, hangs 디렉토리가 생성되어 있다. queue 디렉토리에는 초기 test case를 포함해 퍼징이 진행되며 변형된 입력 파일들이 저장되어 있고, crashes에는 SIGSEGV, SIGILL, SIGABRT 등의 치명적인 오류가 발생한 unique crash에 대한 test case들이 저장되어 있고, hangs에는 프로그램에 타임 아웃이 발생하는 test case가 저장되어 있다. crashes 디렉토리의 파일들을 디버깅하며 exploit을 진행할 수 있다. 여기서 ASAN이라는 기능을 사용하면 크래시 파일에 대한 정보를 매우 자세하게 얻을 수 있다.

* **ASAN**

ASAN은 Address Sanitizer의 약자로 메모리 오류를 탐지하는 도구이다. 버퍼 오버플로우, UAF, 메모리 누수 등을 탐지하는 기능을 갖고 있다. ASAN을 사용하기 위해 dact에 ASAN instrumentation 코드를 삽입해 빌드해야한다. 이전에 AFL instrumentation을 삽입해 빌드한 dact-0.8.42 디렉토리로 이동해 다음의 명령어를 실행한다.

$ make clean

$ CC=”clang -fsanitize=address” CXX=”clang++ -fsanitize=address” ./configure

$ make

이렇게 빌드된 dact를 이용해 /output/crashes의 파일들을 압축해제하면 다음과 같은 결과를 얻을 수 있다.

텍스트, 스크린샷, 메뉴이(가) 표시된 사진

자동 생성된 설명

해당 크래시는 힙 BOF를 발생시키고, 어느 메모리 주소에서 버퍼 오버플로우가 발생하는지 확인할 수 있다. crashes 디렉토리의 모든 파일에 대해 sanitize를 진행하기 위해 다음과 같은 san.sh 쉘 스크립트를 작성하고 crashes 디렉토리에서 실행한다.

for file in \*

do

echo input : $file >> crash.log

../../dact\_ASAN -dcf $file 2>> crash.log

done

이후 생성된 crash.log 파일에서 grep을 이용해 스택 버퍼 오버플로우가 발생한 경우만 확인한다.

텍스트, 스크린샷, 폰트, 정보이(가) 표시된 사진

자동 생성된 설명

총 5개의 스택 BOF를 발생시키는 크래시 파일이 존재한다. crash.log 파일을 열어 각 크래시 파일의 id를 찾아보면,

id:000011,sig:11,src:000000,op:havoc,rep:32

id:000023,sig:11,src:000030,op:havoc,rep:8

id:000031,sig:11,src:000155,op:flip1,pos:26

id:000032,sig:06,src:000155,op:flip2,pos:26

id:000034,sig:06,src:000155,op:havoc,rep:4

위와 같다. 여기서 첫 번째 파일을 exploit에 사용할 것이다. 작업의 편의성을 위해 파일의 이름을 targetcrash로 변경하였다. targetcrash를 dact\_ASAN으로 다시 취약점을 트리거해보고 결과를 분석해본다.

Address 0x7ffce662f940 is located in stack of thread T0 at offset 2304 in frame

#0 0x55b175ee2ebf in dact\_process\_file /home/dmskhs0912/afl\_project/dact-0.8.42/dact\_common.c:249

This frame has 16 object(s):

[32, 36) 'cipher.addr'

[48, 192) 'filestats' (line 250)

[256, 2304) 'file\_extd\_urls' (line 252) <== Memory access at offset 2304 overflows this variable

[2432, 2433) 'algo' (line 253)

[2448, 2449) 'ch' (line 254)

[2464, 2472) 'hdr\_buf' (line 255)

[2496, 2499) 'version' (line 257)

[2512, 2513) 'file\_opts' (line 258)

[2528, 2536) 'filesize' (line 260)

[2560, 2564) 'blk\_cnt' (line 261)

[2576, 2580) 'file\_extd\_size' (line 261)

[2592, 2596) 'blksize' (line 261)

[2608, 2612) 'blksize\_uncomp' (line 261)

[2624, 2628) 'magic' (line 262)

[2640, 2644) 'x' (line 265)

[2656, 2664) 'offset' (line 266)

HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork

(longjmp and C++ exceptions \*are\* supported)

SUMMARY: AddressSanitizer: stack-buffer-overflow /home/dmskhs0912/afl\_project/dact-0.8.42/dact\_common.c:478:40 in dact\_process\_file

T0 스레드에서 BOF가 발생했으며 stack frame에 대한 정보를 보여주고 있다. frame에 16개의 오브젝트가 있으며 [256, 2304) ‘file\_extd\_urls’는 256바이트부터 2303 바이트까지 범위의 오브젝트인데, offset 2304에 접근하려 했기 때문에 오버플로우가 발생한 것을 확인할 수 있다. 또, SUMMARY에서 확인할 수 있듯이 dact\_common.c의 478행에 취약점이 존재함을 알 수 있다.