<FR-gnupg1-4-13.c 코드 분석>

1. 전체 코드

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <symbol.h>

#include <fr.h>

#include <util.h>

#define SAMPLES 100000

#define SLOT 2000

#define THRESHOLD 100

char \*monitor[] = {

"mpih-mul.c:85",

"mpih-mul.c:271",

"mpih-div.c:356"

};

int nmonitor = sizeof(monitor)/sizeof(monitor[0]);

void usage(const char \*prog) {

fprintf(stderr, "Usage: %s <gpg-binary>\n", prog);

exit(1);

}

int main(int ac, char \*\*av) {

char \*binary = av[1];

if (binary == NULL)

usage(av[0]);

fr\_t fr = fr\_prepare();

for (int i = 0; i < nmonitor; i++) {

uint64\_t offset = sym\_getsymboloffset(binary, monitor[i]);

if (offset == ~0ULL) {

fprintf(stderr, "Cannot find %s in %s\n", monitor[i], binary);

exit(1);

}

fr\_monitor(fr, map\_offset(binary, offset));

}

uint16\_t \*res = malloc(SAMPLES \* nmonitor \* sizeof(uint16\_t));

for (int i = 0; i < SAMPLES \* nmonitor ; i+= 4096/sizeof(uint16\_t))

res[i] = 1;

fr\_probe(fr, res);

int l = fr\_trace(fr, SAMPLES, res, SLOT, THRESHOLD, 500);

for (int i = 0; i < l; i++) {

for (int j = 0; j < nmonitor; j++)

printf("%d ", res[i \* nmonitor + j]);

putchar('\n');

}

free(res);

fr\_release(fr);

}

1. 함수 분석
2. Monitor[]: gnupg가 연산 시 접근하는 함수 중, mul, reduce. Square를 수행하는 함수 코드 & line을 저장(Square 연산 : "mpih-mul.c:271“, Reduce 연산 : "mpih-mul.c:85", Multiply 연산: "mpih-div.c:356)
3. Getsymboloffset(char\* file, char\* symbol): exe 파일 중 해당 symbol을 가진 line의 offset을 리턴한다

* 4가지 input format을 인식할 수 있다.( file offset, virtual addresse, loader symbol and line number)
* 먼저 수행 파일 중 ‘:’라는 명령이 있다면, 특정 코드의 line을 참조하겠다는 의미 . 없으면 다음 코드로 넘어가는 듯. 있으면 ptr:0, 없으면 ptr: symbol( : 있으면 해당 line될 떄까지 접근함). 없으면 특정 함수의 위치를 찾는가봄..

2-1) symbol\_bfd.h에서 사용되는 함수들의미

* Symbol file: 컴파일 시 링크 과정에서 여러 목적 파일들을 합치는데, 여러 코드들의 위치를 담은 file(디버깅시 유용하다)
* Map file: 함수 주소와 소스코드의 번호를 기록 함
* BFD library: 목적 파일을 일반적인 관점으로 표현하는 라이브러리. 여러 symbol들의 정보를 저장하거나 응용 프로그램에 전달하게 도와주는 library => 여기에 symbol table, header,섹션 등이 있다.
* Symbol: object file이 만들어지고, o 파일이 쓰이는 변수나 함수들의 위치. 이것 각각에 대한 정보를 한 곳에 모아둔 것이 symtab 즉 symbol table이다.
* 즉, 심볼은: obg file에서 쓰이는 함수나 변수 정보, 그 정보들을 모은 것이 symbol table
* 그래서 실행파일일 특정 function을 call 하면 최종 주소를 obj file 내부의 symbol table을 찾아가서 위치를 조회 후 찾아간다.

2-2) symbol\_bfd 함수 명령

* Char\* file : .exe file임. Binary file descriptor의 형식으로 해당 file을 오픈. 그 file 내부에 있는 symbol table의 크기를 알아내서(시스템 함수 사용) 메모리에 할당 하고, symbol table의 정보를 복사해서 symbol table의 name 중 내가 찾고자 하는 name(함수나 변수)를 찾으면, symbol table에서 해당 변수의 시작 offset & file의 위치 리턴함
* 만약 symbol table을 안쓰고, address를 가지고 있다면, 해당 .exe(혹은 .obj) 파일을 bfd\* 형식으로 열어서 해당 bfd 속에 있는 section을 접근. For loop로 해당 dfd의 section을 순회하면서 찾는다. (VMA: virtual memory area,
* Offsest: 입력받은 address – VMA + 해당 section 의 시작 주소

1. fr\_monitor(fr, map\_offset(binary, offset))

:위에서 찾은 offset을 통해해당 file + offset을 자신의 memory로 할당한 후 fr에 삽입한다.

* mmaps함수를 통하면 자동으로 file에 대한 새로운 주소 매핑 + 거기다가 offset만 추가하면 monitor의 위치를 가져올 수 있다.

1. Fr\_probe(fr, res): 여기서 res의 개수를 최대 10000개로 지정함

* Fr-> VL(내가 모니터링 하고자 하는 line들의 위치)를 memaccess할 때 걸렸던 시간을 res에 저장. 후 clflush//
* 그럼 결론적으로 res[100000]에는 monitor[0]접근 시간, monitor[1]접근 시간, monitor[2]접근 시간하고 끝남( 맨 처음의 probe는)

1. Fr\_trace(fr, sample개수, res,slot,threshold, 500) {

* Slot 시간 만큼 probe 실행해서 rest[0~3]에 계속 갱신
* Slot 시간일 때 is\_active인지 확인해서 cache에 접근한 적이 있었는지 확인
* 최초 hit이 발생한 순간부터 본격적인 trace 시작
* Result += 3 해서 res[]배열의 다음3칸에 fr\_probe한 값을 저장 후 is active 확인. 만약 slot기간 내에 접근을 안햇으면 idle\_count= 0으로 다시 초기화함

Ide\_count:연속적으로 n개의 slot시간동안 접근을 안했단 소리임. 이 값이 500이 되면 계속 접근안한다는 소리이므로 trace 종료

-만약 현재시간 > slot으로 추적 놓쳤으면 방금 적은 부분을 삭제

-idle\_max만큼 도달했거나 res가 꽉 찼으면 count 횟수 return

1. FR-gnupg의 최종 출력값

-위에서 구한 count 횟수만큼 loop 생성(한 slot 당 접근한 것들)

- for(저장된 slot 횟수,i) {

For(각 slot당 연속으로 3개씩 접근시간 저장,j) {

Printf(“i\*3 + j 값)

}