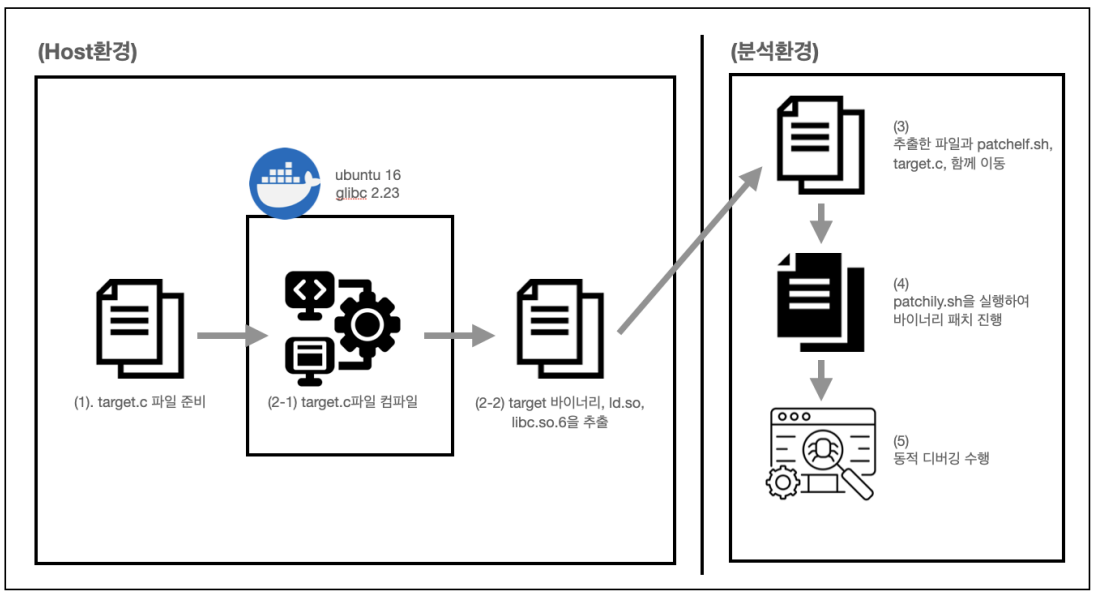
LegacyDynEnv Project(Legacy Dynamic Environment Project)

(github링크)
Top

```
[시스템 구축하고자한 이유]
glibc 2.23버전을 쓰기위해서는 ubuntu16버전이 필요하다.
그래서 도커에서 ubuntu16버전을 구축한 후 내부에서 분석을 진행하려 했지만 파이썬 3.6까지만 지원한다던가 gdb플러그인이 작동이 정상적으로 안되어 분석하는데 제약사항이 존재했다.

따라서 오히려 역으로 최신 우분투에서 glibc2.23버전을 사용하는 바이너리를 동적 디버깅이 가능하도록 하면 어떨까하는 생각에 이 시스템 구축하고자 생각했다.
```

시스템 구성



(그림1 시스템 구성도)

---

1. 동적 디버깅 환경 구축 실험 과정

```
1. LD_PRELOAD, LD_LIBRARY_PATH를 이용한 동적 링킹 수정
2. glibc2.23버전의 libc.so.6, ld.so를 다운 받아서 pwntools이용
3. 1번과정 + patchelf + 16버전에서 컴파일된 바이너리 이용
```

(1) LD_PRELOAD, LD_LIBRARY_PATH를 이용한 동적 링킹 수정
=> 후킹으로

(2) glibc2.23버전의 libc.so.6, ld.so를 다운 받아서 pwntools이용
=> 인터넷 검색을 통해 pwntools코드로 외부 ld.so, libc.so.6을 적용시키는 방법을 확인
하지만 elf바이너리에 glibc버전에 명시되어있어 링킹된 libc.so.6에는 해당 버전이 not found라고 표시됨.

```python
# test.py
from pwn import *

context.terminal=['tmux', 'splitw', '-h']
context.log_level = 'debug'

filename = "./target"
p = process(["./2.23/ld-2.23.so", filename], env={"LD_PRELOAD": "./2.23/libc-2.23.so"})
#p = process(filename)
#gdb.attach(p)

raw_input("test")

p.recvall()
```

```
Received 0x55 bytes:
    b"./target: ./2.23/libc-2.23.so: version `GLIBC_2.34' not found (required by ./target)\n"
```

(문제사항)
=> gdb.attach - pwntools로 gdb attach 하기엔 ptrace 권한 문제가 있음

(3) 1번과정 + patchelf + 16버전에서 컴파일된 바이너리 이용

(왜 2번방법에서 에러가 발생 했을까 이유 생각)

```
strings ./2.23/libc-2.23.so | grep GLIBC
```

=> GLIBC_2.2.5 - GLIBC2.23까지 명시

```
strings /lib/x86_64-linux-gnu/libc.so.6 | grep GLIBC
```

=> GLIBC_2.2.5 - GLIBC2.35까지 명시

따라서 최신 ubuntu에서 컴파일하면 컴파일할때 glibc버전이 포함되어서 libc-2.23.so로 로딩시도하며
명시된 GLIBC2.34가 not found로 뜨는거라 추측

-> 그래서 생각한 다음 방법

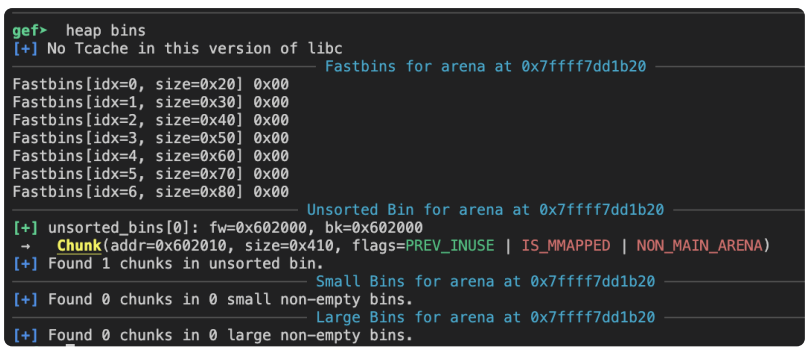
- target바이너리에 존재하는 GLIBC_2.34명시되어있는 부분 변경
- 컴파일할때 외부 libc.so.6으로 링킹이 가능한가?
  => 원래 gcc컴파일 할 때 glibc2.23버전으로 바꿔서 진행할까 햇지만 실패
  (상위 우분투 - 하위 libc버전)

-> 또다른 생각한 방법

- 최신 우분투에서 컴파일해서 최신 libc가 적용되는거라면 구버전에서 컴파일한 바너리를 가져다가 patchelf 이용하면 glibc "not found"가 안뜨고 잘 작동하지 않을까 예상

환경 구축 결과

=> 최신 ubuntu버전에서 위 과정을 통해 얻은 (컴파일된 바이너리, ld.so, libc.so.6)을 동적 디버깅 시도



(그림2 target바이너리 gdb결과)

해당 바이너리는 malloc, free함수를 이용하는 구문이 있다.
(그림2)은 free함수를 실행한 이후에 상황을 넣은 것인데 tcache가 아닌 unsorted bin에 들어간 모습을 봐서 구 버전의 라이브러리가 실행된 것을 알 수 있다.

(바이너리 추가 확인)

```
root@069442a802fd:~/test# ldd test
        linux-vdso.so.1 (0x00007fffd5dcb000)
        ./2.23/libc.so.6 (0x00007fae78200000)
        ./2.23/ld.so => /lib64/ld-linux-x86-64.so.2 (0x00007fae78683000)

root@069442a802fd:~/test# strings test | grep GLIBC
GLIBC_2.2.5
free@@GLIBC_2.2.5
printf@@GLIBC_2.2.5
close@@GLIBC_2.2.5
read@@GLIBC_2.2.5
__libc_start_main@@GLIBC_2.2.5
malloc@@GLIBC_2.2.5
open@@GLIBC_2.2.5
perror@@GLIBC_2.2.5
```

=> 동적 링킹(ldd) 및 내부 버전(strings)도 구버전으로 잘 잡혀있는 것을 확인 할 수 있다.

이제 환경 구축을 실험했던 과정을 시스템으로 구축하여 어느정도 자동화한다면
최신 우분투에서도 glibc2.23을 돌릴 수 있지 않을까 생각
-> 단, 반자동화를 택한 이유는 파일을 추출하는데 도커를 써서 이중 도커가 안되고, 사람마다 분석하는 환경이 다르기 때문에 파일 옮겨서 사용하는 부분은 수동으로 하게 만들었다.

2. 반자동화 시스템 구축

```
(시스템 구성)
[host]
1. 동적 디버깅 해야할 코드를 준비(target.c)
```

```
2. ubuntu 16버전 docker를 이용해서 코드를 컴파일하고 내부에서 libc.so.6, ld.so파일을 확보
   (target, libc.so.6, ld.so 파일 확보)
   => low_compile.sh파일 실행

3. 분석하는 환경에 필요한 파일들을 직접 옮김
   (target, libc.so.6, ld.so, target.c, patchelf.sh)
   (수동으로 한 이유 : vmware, docker, local, mac등 다양한 상황을 전부 충족시키기 어려워서)
   => 직접 옮김

[분석환경]
4. patchelf.sh를 실행시켜 바이너리에 대한 패치 진행
   => 내부 코드 중 ldd, strings를 통해 패치 확인

5. "gdb target"을 통해 동적 디버깅 수행
```

(1) 동적 디버깅 해야할 코드 준비
=> target.c 예시 코드

```c
// ex) target.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 1024

int main() {
    char *buffer;
    ssize_t bytesRead;
    int fd;

    // 파일 열기
    fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("파일 열기 실패");
        return EXIT_FAILURE;
    }

    // 메모리 동적 할당
    buffer = (char *)malloc(BUFFER_SIZE * sizeof(char));
    if (buffer == NULL) {
        perror("메모리 할당 실패");
        close(fd);
        return EXIT_FAILURE;
    }

    // 파일에서 데이터 읽기
    bytesRead = read(fd, buffer, BUFFER_SIZE);
    if (bytesRead == -1) {
        perror("읽기 실패");
        free(buffer);
        close(fd);
        return EXIT_FAILURE;
    }

    // 읽은 데이터 화면에 출력
    printf("읽은 데이터: %s\n", buffer);

    // 동적으로 할당된 메모리 해제
    free(buffer);

    // 파일 닫기
    close(fd);

    return EXIT_SUCCESS;
}
```

=> chatgpt에게 malloc, free함수 이용하고 데이터 입력받는 코드 알려달라고 했더니 이것을 알려주었다.

(2) ubuntu 16버전 docker를 이용해서 코드를 컴파일하고 내부에서 libc.so.6, ld.so파일과 함께 확보
=> low_compile.sh파일 실행 (target, libc.so.6, ld.so 파일 확보)

```bash
# low_compile.sh
#! /bin/bash

# target file name(args)
file_name="target.c"
folder_path="./export" # 확인할 폴더 경로

# 폴더가 존재하는지 확인
if [ ! -d "$folder_path" ]; then
    mkdir -p "$folder_path"
    echo "[!] export 폴더가 생성되었습니다."
fi

echo "[*] copy internal file"
cp -r $file_name ./export/$file_name
cp -r patchelf.sh ./export/patchelf.sh

echo "[*] Build container image"
docker build --build-arg file=$file_name -t img_low_compile . # build명령어
```

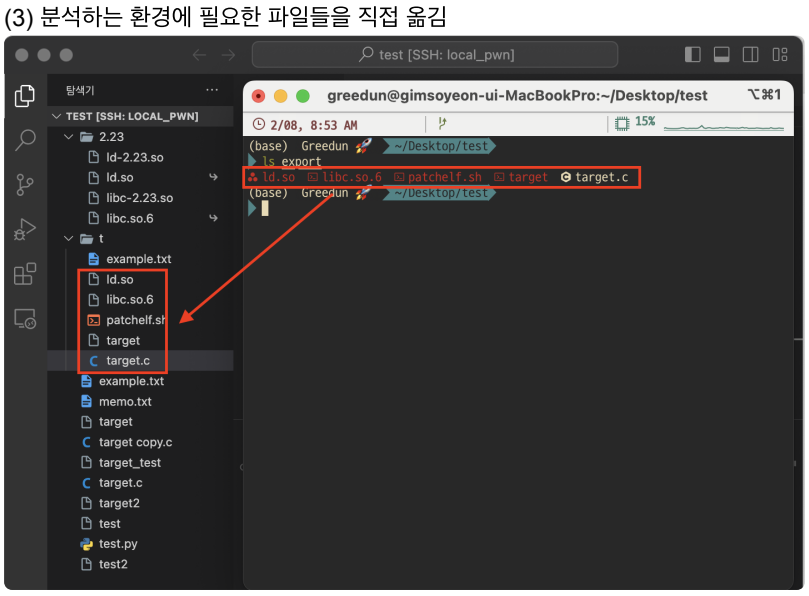
```
echo "[*] Run container"
docker run --name low_compile -d img_low_compile # run명령어(백그라운드)
# docker run --name compile -it img_low_compile /bin/bash # run명령어(접속용)

# 파일 꺼내오기
echo "[*] export file(target , libc-2.23.so , ld-2.23.so)"
docker cp low_compile:/target ./export/target
docker cp low_compile:/lib/x86_64-linux-gnu/libc-2.23.so ./export/libc.so.6
docker cp low_compile:/lib/x86_64-linux-gnu/ld-2.23.so ./export/ld.so

# 컨테이너 종료
echo "[*] remove container"
docker rm -f low_compile

# export 결과
echo "[*] result export file"
ls ./export
```

(3) 분석하는 환경에 필요한 파일들을 직접 옮김



(그림3 export폴더에 있는 파일들을 분석환경으로 복사)

(4) patchelf.sh를 실행시켜 바이너리 패치 진행

```
# patchelf.sh
# chmod +x patchelf.sh # 외부에서 진행

chmod +x target
chmod +x ld.so
chmod +x libc.so.6

patchelf --set-interpreter ./ld.so target
patchelf --replace-needed libc.so.6 ./libc.so.6 target

# print
echo "> ldd target"
ldd target
echo ""
echo "> strings target | grep GLIBC"
strings target | grep GLIBC
```

=> 바이너리 종속되어 있는 ld.so와 libc.so.6을 patchelf를 이용하여 변경해준다.

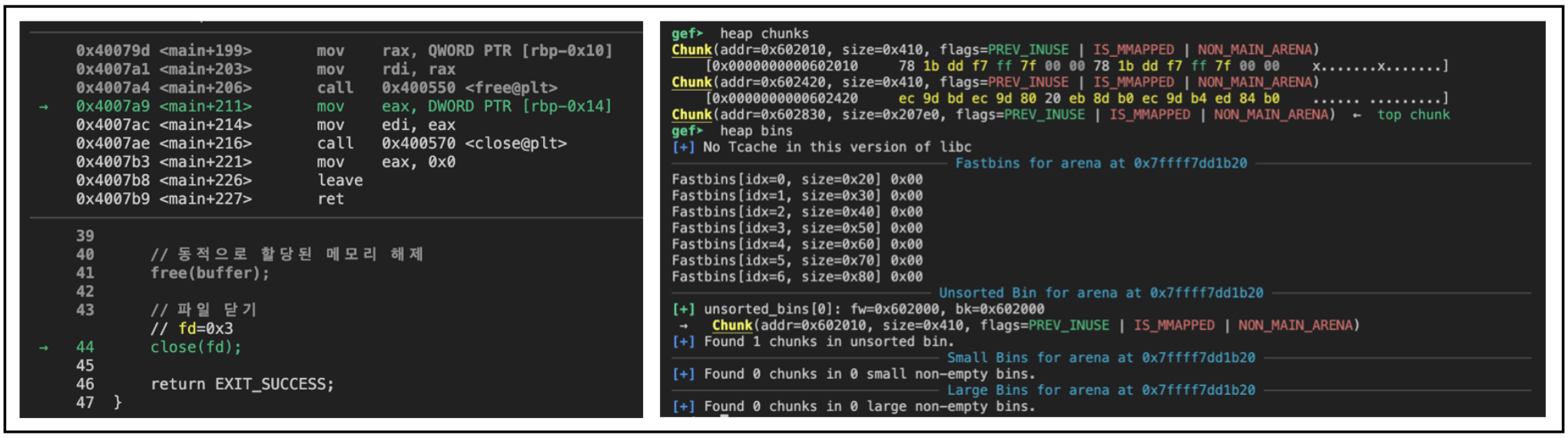
(그림4 바이너리 패치 전후 비교)

(그림4) 좌측은 구버전 우분투 컴파일에 바이너리 패치를 한 사진이고, 우측은 일반적인 컴파일과 라이브러리 맵핑을 보여준다.
차이점은 이전버전에서는 GLIBC버전이 2.2.5만 있는 반면 최신버전에는 2.2.5뿐만 아니라 2.34가 같이 존재한다.
또한 라이브러리 맵핑도 좌측은 현재폴더에 있는 ld.so, libc.so.6을 가리키지만 우측은 /lib폴더에 있는 파일들을 가리킨다.

(5) "gdb target"을 통해 동적 디버깅 수행

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x00000000003fe000 0x00000000003ff000 0x0000000000000000 rw- /root/test/t/target
0x0000000000400000 0x0000000000401000 0x0000000000002000 r-x /root/test/t/target
0x0000000000600000 0x0000000000601000 0x0000000000002000 r-- /root/test/t/target
0x0000000000601000 0x0000000000602000 0x0000000000003000 rw- /root/test/t/target
0x00007ffff7a0d000 0x00007ffff7bcd000 0x0000000000000000 r-x /root/test/t/libc.so.6
0x00007ffff7bcd000 0x00007ffff7dcd000 0x00000000001c0000 --- /root/test/t/libc.so.6
0x00007ffff7dcd000 0x00007ffff7dd1000 0x00000000001c0000 r-- /root/test/t/libc.so.6
0x00007ffff7dd1000 0x00007ffff7dd3000 0x00000000001c4000 rw- /root/test/t/libc.so.6
0x00007ffff7dd3000 0x00007ffff7dd7000 0x0000000000000000 rw-
0x00007ffff7dd7000 0x00007ffff7dfd000 0x0000000000000000 r-x /root/test/t/ld.so
0x00007ffff7ff3000 0x00007ffff7ff6000 0x0000000000000000 rw-
0x00007ffff7ff6000 0x00007ffff7ffa000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000025000 r-- /root/test/t/ld.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x0000000000026000 rw- /root/test/t/ld.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

(그림5 vmmap명령어로 패치된 바이너리 메모리맵 확인)



(그림6 context화면과 heap구조 확인)

vmmap으로 ld.so, libc.so.6의 맵핑 상황을 확인하고 동적디버깅을 해보았다.
이 시스템을 구축하려던 목적이 heap에서 glibc2.23이 적용시키지 위해서이기 때문에 free함수까지 진행시켰다.
free함수까지 지났을때 (그림6) 우측사진에 heap상황을 보니 tcache는 해당 libc버전에 없다고 명시되고 unsorted bin에 해제된 청크가 들어간 것을 확인하였다.

3. 이후에 보완할 점 명시

원래 구상한 시스템은 다른 버전의 libc을 libc,ld파일만 있다면 자유롭게 패치하도록 하는 것이 목적이었다.
하지만 바이너리를 지정 libc파일로 컴파일하는 것이 실패한다던지, LD_PRELOAD로 하는 것이 실패하여
구버전 우분투의 도커에서 컴파일하여 가져오고 이용하는 형태로 진행했다. 이 부분은 도커 ubuntu의 libc, ld버전으로 한정된다는 한계점이 존재한다.

이 한계점을 보완하기 위해서 생각한 방법은

- ubuntu16버전만 호환성이 깨진다는 것을 증명하여 다른 libc버전은 patchelf로 스크립트화
- 컴파일할 때 라이브러리 지정하여 가능하게 하기
- ...
  일단은 명시만 해두고 나중에 필요할 때 보완하도록 하겠다.