[문제 정보]

이름	HolyAutoTapper
분야/난이도	리버스 엔지니어링 (난이도 : 중)
문제풀이 기법	➢ 바이너리 디핑(diffing) ➢ Frida 후킹
권장 풀이 환경	➢ MuMu Player, LD Player 등의 루팅된 상태의 에뮬레이터 환경
FLAG	649e4bc821f83e57302c31f7e9cd3384423a65d35c142ac0adcd81fe13f22e80

[문제 풀이 방향성]

앱은 루팅된 안드로이드 환경(실제 휴대폰, 에뮬레이터 등)에서 실행할 수 있고, 앱에서 실제 매크로 기능을 동작시킬 수 있다.

DEX 코드부터 분석해 보면, smali 코드 내부에는 터치를 트리거하는 코드가 존재하지 않는다. Native 영역과의 연결 부분을 살펴보면 init_run 함수가 나오는데, 이 함수에서 터치 좌표와 반복 정보 등을 받아 실질적으로 터치를 수행한다.

```
public final native String init_run(String basePath, String touchPointsJson, int repeatCount, int repeatInterval);
```

그림 1 : dex 파일 내, native 함수 정의문

```
public final Object invokeSuspend(Object obj) {
    Object p2;
    C0061g c0061g = this.f200c;
    HolyShield holyShield = this.f199b;
    W.a.O(obj);
    try {
        HolyShield.b(holyShield);
        String c2 = HolyShield.c(holyShield, c0061g.f190a);
        Object value = holyShield.f868b.getValue();
        kotlin.jvm.internal.n.d(value, "getValue(...)");
        String absolutePath = ((File) value).getAbsolutePath();
        kotlin.jvm.internal.n.d(absolutePath, "getAbsolutePath(...)");
        p2 = holyShield.init_run(absolutePath, c2, c0061g.f191b, c0061g.f192c);
    } catch (Exception e2) {
        p2 = W.a.p(e2);
    }
    return new I.g(p2);
}
```

그림 2 : init_run 함수 호출부

참고로, init_run 을 포함한 Native 함수는 apk 에 포함된 libholy.so 안에 정의되어 있음

```
static {
    System.loadLibrary("holy");
}
```

그림 3 : dex 파일 내 libholy.so 로드 부분

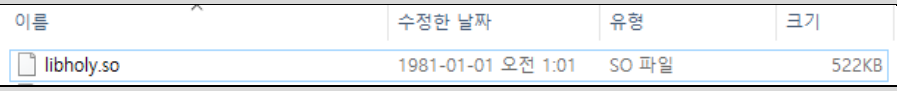


그림 4 : lib/{arch}/libholy.so 파일

libholy.so 를 디컴파일해 보면, 터치와 같은 매크로 동작 로직은 보이지 않는다.

그러나, Lua 관련 라이브러리 함수들이 확인되는 점을 통해, 매크로 동작에 Lua 코드가 관여하는 것으로 추정할 수 있다.

참고로, libholy.so 에서 사용하는 모든 문자열은 정적 분석을 방해하기 위해 특정 해시 배열과 XOR 인코딩된 상태로 존재한다. 실제 실행 시에는 init_array 에 정의된 디코딩 함수를 통해 원본 문자열을 복원한다.

따라서, 정적 분석 시에는 해당 디코딩 함수를 분석하여 원본 문자열들을 파악해 나가는 것이 핵심이다.

```
.init_array:0000000000081A80 _init_array     segment qword public 'DATA' use64
.init_array:0000000000081A80                 assume cs:_init_array
.init_array:0000000000081A80                 ;org 81A80h
.init_array:0000000000081A80                 dq offset sub_31170     ; 바이너리 내, 모든 문자열 복호화 수행
.init_array:0000000000081A80 _init_array     ends
```

그림 5 : .init_array 에 정의된 문자열 복호화 함수

다시 init_run 함수로 돌아와 살펴보면, 함수 시작 부분에서 fopen 을 통해 특정 파일을 여는 것을 확인할 수 있다. 이 파일의 경로는 /data/data/com.sample.holyautotapper/files/touch.lua 이다.

```
v18 = std::string::append(&dest, "/");
ptr = *(v18 + 16);
v60 = *v18;
*v18 = 0;
*(v18 + 16) = 0;
v19 = std::string::append(&v60, off_839A0);    // touch.lua
v63 = *(v19 + 16);
*filename = *v19;
*v19 = 0;
*(v19 + 16) = 0;
if ( (v60 & 1) != 0 )
  operator delete(ptr);
if ( (dest & 1) != 0 )
  operator delete(v59);
(*(*a1 + 1360LL))(a1, a3, v12);
if ( (filename[0] & 1) != 0 )
  v20 = v63;
else
  v20 = &filename[1];
v21 = fopen(v20, "rb");                        // lua 파일 열기
if ( !v21 )
{
  std::operator+<char>(&v60, "error: failed to open file: ", filename);
```

그림 6 : touch.lua 파일 로드

이름	수정한 날짜	유형	크기
dexopt	2025-12-09 오전 12:14	파일 폴더	
touch.lua	1981-01-01 오전 1:01	Lua 원본 파일	4KB

그림 7 : assets/touch.lua 파일

이후, Lua 파일 데이터를 읽고 실행을 위해 luaL_loadbufferx 함수에 해당 데이터를 넘긴다.

```
if ( !luaL_loadbufferx(v27, v24, v23, off_839A0, "b") )
{
  v41 = lua_pcallk(v27, 0, 1, 0, 0, 0);
  v60 = 0;
  ptr = 0;
  if ( v41 )
  {
    v42 = lua_tolstring(v27, 0xFFFFFFFFLL, 0);
    v43 = "unknown";
    if ( v42 )
      v43 = v42;
    std::string::basic_string<decltype(nullptr)>(&dest, v43);
    v44 = std::string::insert(&dest, 0, "exec error: ");
```

그림 8 : touch.lua 데이터를 luaL_loadbufferx 함수로 로드하는 로직

luaL_loadbufferx 함수는 내부적으로 lua_load 를 호출하여 Lua 소스를 로드한다.
이 과정에서 lua_load 의 두 번째 인자인 콜백 함수가 호출되며, 해당 콜백에서 소스 데이터의 무결성 검증이 이루어진다.

```
if ( v10 && v11 && v12 )
{
  v17 = (lua_load)(a1, sub_37F30, &v19, a4, a5, v15);
  free(ptr);
  free(v25);
  free(v26);
}
```

그림 9 : lua_load 함수 수행 로직

두 번째 인자의 콜백 함수는 원본 Lua 소스에서 getS 이며, 무결성 검증 외에는 별도의 로직이 존재하지 않는다.

다만, libholy.so 에서는 getS 에 복호화 로직을 추가하여 touch.lua 에 적용된 암호화를 해제하면서 Lua 스크립트를 메모리에 로드하도록 커스터마이징 되어있다.

이 때, luaL_loadbufferx 에서 복호화를 위한 필드 초기화가 진행되기 때문에, 해당 함수에서부터 분석을 시작해야 한다. 실제로 Lua 5.3 소스를 빌드하여 바이너리 디핑(diffing)을 수행해 보면, 신뢰도(Confidence)가 높은 Lua 함수들 중 유독 유사도(Similarity)가 낮게 나타나는 함수가 바로 luaL_loadbufferx 임을 확인할 수 있다.

참고로, lua 버전은 touch.lua 파일의 헤더를 통해 파악할 수 있다 (LUA 뒤에 0x53 이 붙는 구조)

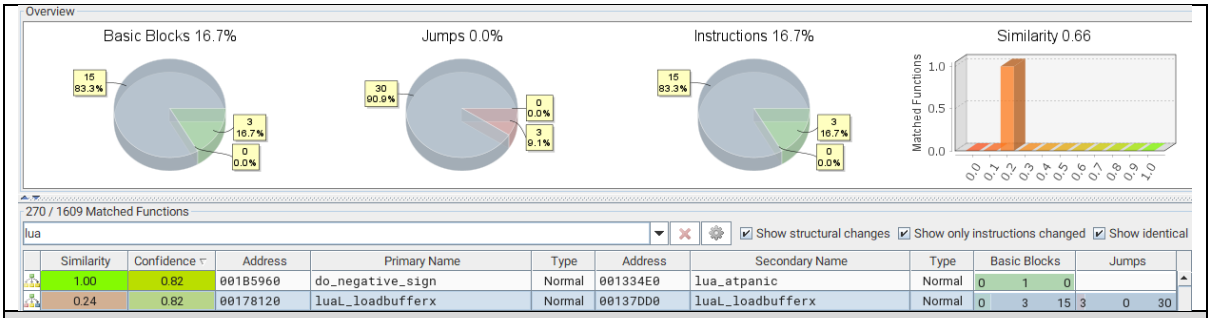


그림 10 : luaL_loadbufferx 원본과의 디핑(diffing) 결과

이 점을 토대로, 실제 lua5.3 소스와 디컴파일 결과를 비교해보면서 분석을 진행할 수 있다.

```
static const char *getS (lua_State *L, void *ud, size_t *size) {
  LoadS *ls = (LoadS *)ud;
  (void)L;  /* not used */
  if (ls->size == 0) return NULL;
  *size = ls->size;
  ls->size = 0;
  return ls->s;
}
```

그림 11 : 원본 getS 소스 github

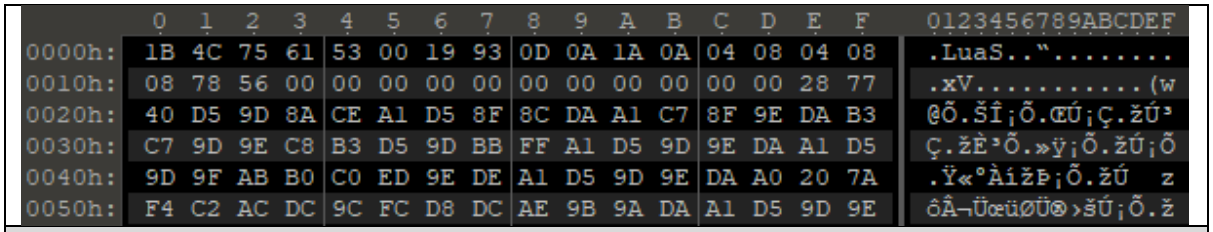
```
v3 = *(a2 + 8);                         // ls->size
if ( !v3 )
  return 0;
*a3 = v3;                               // 여기서부터 새로운 로직 시작
v5 = *(a2 + 24);
if ( !v5 )
  return 0;
v6 = *a2;
if ( !*a2 )
  return 0;
v68 = v3;
memcpy(v5, v6, v3);
```

그림 12 : 커스텀 getS 소스 일부

커스텀 getS 함수는 크게 아래와 같은 로직으로 구성되어 있다. 해당 복호화 로직을 Python 으로 구현하면 touch.lua 를 복호화할 수 있다. (※ 해설 및 도구 폴더의 decrpyt_tool.py 참고)

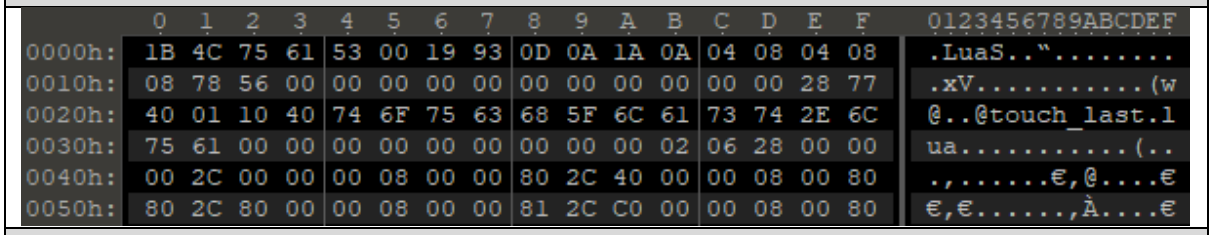
[복호화 과정 (1024 바이트 청크 단위로 수행)]

1. 청크를 1 바이트 단위로 순회하며, "a%^*b" 키 배열과 XOR 연산 후 NOT 연산을 수행한다.
2. (1)에서 처리된 데이터의 각 바이트에 대해 특정 state 값과 XOR 연산을 수행하고, 그 결과를 순차적으로 누적한다.
3. (2)까지 처리된 데이터를 역순으로 재배치하면 평문 데이터가 된다.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	1B	4C	75	61	53	00	19	93	0D	0A	1A	0A	04	08	04	08	.LuaS..“.....
0010h:	08	78	56	00	00	00	00	00	00	00	00	00	00	00	28	77	.xV.....(w
0020h:	40	D5	9D	8A	CE	A1	D5	8F	8C	DA	A1	C7	8F	9E	DA	B3	@Õ.ŠÎ¡Õ.ŒÚ¡Ç.žÚ³
0030h:	C7	9D	9E	C8	B3	D5	9D	BB	FF	A1	D5	9D	9E	DA	A1	D5	Ç.žÈ³Õ.»ÿ¡Õ.žÚ¡Õ
0040h:	9D	9F	AB	B0	C0	ED	9E	DE	A1	D5	9D	9E	DA	A0	20	7A	.Ÿ«°ÀížÞ¡Õ.žÚ z
0050h:	F4	C2	AC	DC	9C	FC	D8	DC	AE	9B	9A	DA	A1	D5	9D	9E	ôÂ¬Üœüø Ü®›šÚ¡Õ.ž

그림 13 : 암호화 상태의 touch.lua



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	1B	4C	75	61	53	00	19	93	0D	0A	1A	0A	04	08	04	08	.LuaS..“.....
0010h:	08	78	56	00	00	00	00	00	00	00	00	00	00	00	28	77	.xV.....(w
0020h:	40	01	10	40	74	6F	75	63	68	5F	6C	61	73	74	2E	6C	@..@touch_last.l
0030h:	75	61	00	00	00	00	00	00	00	00	00	02	06	28	00	00	ua.....(..
0040h:	00	2C	00	00	00	08	00	00	80	2C	40	00	00	08	00	80	.,.....€,@....€
0050h:	80	2C	80	00	00	08	00	00	81	2C	C0	00	00	08	00	80	€,€.....,À....€

그림 14 : 복호화 상태의 touch.lua

복호화 결과, 스크립트 문자열이 정상적으로 확인된다. 그러나 이 파일은 순수 Lua 스크립트가 아닌, 컴파일된 luac 바이트코드 파일이다.

luadec(링크)을 사용하여 디컴파일하면 원본 Lua 스크립트를 복원할 수 있다.
복원된 스크립트 내부에는 매크로 동작 함수와 플래그 생성 함수가 정의되어 있으며, 플래그 생성 함수를 실행하면 최종 플래그를 획득할 수 있다.

```
box@ubuntu:~/Desktop/luadec/luadec$ ./luadec touch.lua.enc
-- Decompiled using luadec 2.2 rev: 895d923 for Lua 5.3 from https://github.com/viruscamp/luadec
-- Command line: touch.lua.enc

processing OP_JMP to } else {
 at line 2649 in file decompile.c
 for lua files: touch.lua.enc
 at lua function 0_3 pc=27

processing OP_JMP to } else {
 at line 2649 in file decompile.c
 for lua files: touch.lua.enc
 at lua function 0 pc=36

-- params : ...
-- function num : 0 , upvalues : _ENV
tap = function(x, y)
  -- function num : 0_0 , upvalues : _ENV
  local cmd = (string.format)("input tap %d %d", x, y)
  ·
```

그림 15 : luadec 을 통한 luac 디컴파일

```
get_flag = function()
  -- function num : 0_4 , upvalues : _ENV
  local chars = "abcdefghijklmnopqrstuvwxyz0123456789"
  local indices = {33, 31, 36, 5, 31, 2, 3, 35, 29, 28, 6, 35, 30, 5, 32, 34, 30, 27, 29, 3, 30, 28,
  local flag = ""
  for i = 1, #indices do
    local idx = indices[i]
    flag = flag .. (string.sub)(chars, idx, idx)
  end
  return flag
end
```

그림 16 : flag 획득 함수

추가로, Lua 동작 구조를 잘 이해하고 있다면, 복호화 알고리즘을 몰라도 frida 를 통해 getS 리턴 값만 잘 덤프하면 touch.lua 확보가 가능하다. (※ 해설 및 도구 폴더의 dump_decrypt_lua.js 참고)