

前言

前面介绍了通过静态读代码的方式去发现问题，这里介绍两种 `fuzz` 目标软件的方式。

相关文件

链接: <https://pan.baidu.com/s/1l6BuuL-HPFdkFSVNOLpjUQ>
提取码: erml

使用winaf1

`winaf1` 是 `af1` 在 `windows` 上的移植版本，这里首先尝试使用 `winaf1` 去 `fuzz` 一下目标软件中感兴趣的函数。为了学习 `winaf1` 的使用，可以先写一个 `demo` 程序来学习一下 `winaf1` 的用法。

示例

为了做示范，这里我们开发一个 `d11`，这个 `d11` 会导出一个函数 `vuln`。

```
__cdecl vuln(char *a1)
{
    char v2; // [esp+0h] [ebp-E0h]
    char v3; // [esp+1h] [ebp-DFh]
    char v4; // [esp+C8h] [ebp-18h]
    FILE *v5; // [esp+DCh] [ebp-4h]

    v5 = fopen(a1, "rb");
    if ( v5 )
    {
        fread(&v2, 0xC8u, 1u, v5);
        fclose(v5);
    }
    if ( v2 == -56 && v3 == -56 )
        memcpy(&v4, &v2, 0xC8u);
    return 1;
}
```

函数的功能比较简单，参数是文件路径，程序首先打开文件然后读取内容。当文件的开头 2 个字节均为 `\xc8` 时就会触发栈溢出。

下面我们看看怎么用 `winaf1` 来 `fuzz` 它。

`winaf1` 不能直接去 `fuzz` 动态链接库，所以首先需要写一个加载程序，加载程序的逻辑如下。

- 首先加载 `d11`
- 获取要 `fuzz` 的目标函数在内存中的地址

- 把 `winafl` 生成的样本数据交给目标函数进行处理。

具体代码如下

```
// 读取文件内容
typedef int (*vuln) (char* path);
vuln pvuln = NULL;

int fuzzme(char* fpath){
    return pvuln(fpath);
}

int _tmain(int argc, _TCHAR* argv[])
{
    HMODULE hMod = LoadLibrary(_T("vuln.dll")); //dll路径
    if (hMod)
    {
        pvuln = (vuln)GetProcAddress(hMod, "vuln");
        int ret = fuzzme(argv[1]);
        printf("%d", ret);
        FreeLibrary(hMod);
    }else
    {
        MessageBox(NULL, TEXT("vuln.dll 模块加载失败"), TEXT("警告"), 0);
        exit(0);
    }
    return 0;
}
```

就把命令行的第一个参数作为文件路径传入 `vuln` 函数。

下面用 `winafl` 测试一波

```
af-fuzz.exe -i F:\security_tools\winafl\fuzz\demo\in\ -o
F:\security_tools\winafl\fuzz\demo\out -D F:\security_tools\winafl\dynamorio\bin32 -t 20000
-- -coverage_module vuln.dll -target_module fuzz.exe -target_offset 0x1000 -nargs 1 --
F:\security_tools\winafl\fuzz\demo\fuzz\Release\fuzz.exe @@
```

其中

```
-target_module 加载程序名
-target_offset 为 fuzzme 的函数偏移
-D 指定 dynamorio 的路径
-coverage_module 指定 winafl 统计代码覆盖了的模块，一般为待测函数所在的模块
@@ 是占位符，winafl会把生成的样本文件的路径替换掉 @@
```

具体参数的意思可以看官方文档。

这里就表示每次 `fuzz` 生成的样本文件的路径在命令行的第一个参数里面，然后在程序中我们把第一个参数传给了漏洞函数（因为目标函数要的参数就是一个文件路径），这样我们就能对 `vuln` 函数进行 `fuzz`。

fuzz 截图

```
WinAFL 1.14 based on AFL 2.43b (fuzz.exe)

+- process timing -----+ overall results -----+
|   run time : 0 days, 0 hrs, 0 min, 33 sec   | cycles done : 33   |
|   last new path : 0 days, 0 hrs, 0 min, 29 sec   | total paths : 2   |
| last uniq crash : 0 days, 0 hrs, 0 min, 29 sec   | uniq crashes : 1   |
|   last uniq hang : none seen yet               |   uniq hangs : 0   |
+- cycle progress -----+ map coverage -----+
| now processing : 1 (50.00%)                   | map density : 0.01% / 0.01%   |
| paths timed out : 0 (0.00%)                   | count coverage : 1.00 bits/tuple   |
+- stage progress -----+ findings in depth -----+
| now trying : havoc                             | favored paths : 1 (50.00%)   |
| stage execs : 32/307 (10.42%)                 | new edges on : 2 (100.00%)   |
| total execs : 3023                             | total crashes : 23 (1 unique)   |
| exec speed : 12.94/sec (zzzz...)              | total tmouts : 0 (0 unique)   |
+- fuzzing strategy yields -----+ path geometry -----+
| bit flips : 0/48, 0/46, 0/42                 | levels : 2                 |
| byte flips : 0/6, 0/4, 0/1                   | pending : 0                 |
| arithmetics : 0/335, 0/95, 0/0                | pend fav : 0                 |
| known ints : 0/34, 0/131, 0/40                | own finds : 1                 |
| dictionary : 0/0, 0/0, 0/0                    | imported : n/a                 |
|   havoc : 2/2174, 0/0                        | stability : 100.00%           |
|   trim : 99.99%/18, 0.00%                    |                               |
+-----+-----+
^C-----+ [cpu: 0%]
```

ZipLib.dll

通过前面对程序处理皮肤文件逻辑的分析，发现程序在处理老格式的皮肤文件时会调用程序目录下的 `ZipLib.dll` 来进行解压，我们可以尝试用 `winaf1` 来 `fuzz` 一下。

首先看看 `ZipLib.dll` 的导出函数。

StudyPE+ (x86) 1.09 beta 0 --> ZipLib.dll

文件 选项 工具 杂项 帮助

[基本] [区段] [数据表] [导入] [导出] [资源] [重定位] [异常] [.Net] [汇编] [进程]

ImageExportDirectory

ExportTable RVA 000217D0

Charact.	00000000	TimeStamp	5AEADF9F	nName	0002265C
nBase	00000001	NumOfFunctions	0000000A	NumOfNames	0000000A
AddrOfFunc	000225F8	AddrOfNames	00022620	AddrOfNameOrder	00022648

Ord...	RAV	Function Name	☒...	Entry Point
000...	00003760	FreeUnzipBuf	.text	
000...	00003770	PrepareUnzipFile	.text	
000...	00003970	SetZipLevel	.text	
000...	00003950	UnZip2	.text	
000...	00003200	UnZipEx	.text	
000...	00003960	UnZipEx2	.text	
000...	00003710	UnZipFile	.text	
000...	00003430	UnZipFileFromZip_Test	.text	
000...	00002920	ZipFolder	.text	
000...	00002960	ZipFolderEx	.text	

查看导出表信息及导出函数

17:44:58

看到名字基本就大概猜到功能了。然后通过分析输入法处理老版本皮肤格式的处理的逻辑，发现在 0x49F544 会调用 zipLib.dll 的 UnZipFile 这个函数。

```
v11 = WideCharToMultiByte(0x3A8u, 0, v10 + 2, -1, skin_str, 260, 0, 0); // 把 skin.ini 转成了 ascii
if ( v11 )
{
    v12 = v11 - 1;
    if ( v12 >= 260 )
        v12 = 259;
    skin_str[v12] = 0;
}
else
{
    skin_str[0] = 0;
}
UnZipFile_funcptr = v6[7];
if ( UnZipFile_funcptr ) // 取出 unzip 的函数指针，判断
{
    if ( UnZipFile_funcptr(path, skin_str, &result) ) // 位于 path 路径的 zip 文件中的 skin.ini 文件的内容解压出来，内容的指针 保存到 pDecomp
    {
        *p_len = result; // 把 解压的文件内容的指针 通过 pp_** 传到调用方
        v15 = 1;
        goto LABEL_21;
    }
    v16 = malloc_for_obj(a2);
    sprintf_to_obj(v16, L"%s", L"压缩失败"); // 压缩失败
}
else
```

0009E93C decompress_file_from_zip:00 (49F53C)

通过调试，可以得到 UnZipFile 的参数信息。

- 第一个参数为 zip 文件的路径
- 第二个参数为需要解压的文件内容
- 第三个参数为一个结构体指针，用于传出解压的结果

保存解压结构的结构体的定义如下

```
typedef struct{
    int len; // 内容的长度
    char* buf; // 内容的指针
}result;
```

要进行 fuzz，下面还需要写一个小程序，把被测 api 调起来。具体代码如下

```
#include "stdafx.h"
#include <windows.h>

#include <crtdbg.h>

typedef struct{
    int len; // 内容的长度
    char* buf; // 内容的指针
}result;

// 把 path 的zip 文件中的 target 文件的内容解压出来，内容的指针和长度 保存到 res 里面
typedef int (*UnZipFile) (char* path, char* target, result* res);
typedef int (*FreeUnzipBuf) (result * res);

UnZipFile pUnZipFile = NULL;
FreeUnzipBuf pFreeUnzipBuf = NULL;
```

```

int fuzzme(char* fpath){
    char* target = "skin.ini";
    result res = {0};
    int ret = pUnZipFile(fpath, target, &res);

    printf("res.buf:0x%p, res.len: 0x%p\n", res.buf, res.len);
    if(res.buf){
        pFreeUnzipBuf(&res);
    }

    return ret;
}

int _tmain(int argc, _TCHAR* argv[])
{
    HMODULE hMod = LoadLibrary(_T("ZipLib.dll")); //dll路径
    if (hMod)
    {
        pUnZipFile = (UnZipFile)GetProcAddress(hMod, "UnZipFile"); //直接使用原工程函数名
        pFreeUnzipBuf = (FreeUnzipBuf)GetProcAddress(hMod, "FreeUnzipBuf");
        int ret = fuzzme(argv[1]);
        printf("%d\n", ret);
        FreeLibrary(hMod);
    }else
    {
        MessageBox(NULL, TEXT("ZipLib.dll 模块加载失败"), TEXT("警告"), 0);
        exit(0);
    }
    return 0;
}

```

主要逻辑为 首先用 `LoadLibrary` 把 `zipLib.dll` 加载起来, 然后获取到被测函数的地址, 然后传参数调用。

```

C:\Users\xinSai\Desktop\fuzz\winaf1\bin32> afl-fuzz.exe -D
C:\Users\xinSai\Desktop\fuzz\dynamorio\bin32 -i C:\Users\xinSai\Desktop\fuzz\sougou\zip -o
C:\Users\xinSai\Desktop\fuzz\sougou\out -t 20000 -- -coverage_module ZipLib.dll -
target_module fuzz.exe -target_offset 0x8010 -nargs 1 --
C:\Users\xinSai\Desktop\fuzz\sougou\fuzz.exe @@

```

```

WinAFL 1.14 based on AFL 2.43b (fuzz.exe)
+- process timing -----+- overall results -----+
|   run time : 3 days, 16 hrs, 55 min, 34 sec   |   cycles done : 0   |
|   last new path : 1 days, 3 hrs, 46 min, 6 sec   |   total paths : 338   |
| last uniq crash : none seen yet               |   uniq crashes : 0   |
|   last uniq hang : 1 days, 16 hrs, 38 min, 28 sec   |   uniq hangs : 4   |
+- cycle progress -----+- map coverage -----+
| now processing : 3 (0.89%)                   |   map density : 1.19% / 2.09%   |
| paths timed out : 0 (0.00%)                 |   count coverage : 3.14 bits/tuple   |
+- stage progress -----+- findings in depth -----+
| now trying : bitflip 1\1                   |   favored paths : 41 (12.13%)   |
| stage execs : 18.4k/287k (6.40%)           |   new edges on : 48 (14.20%)   |
| total execs : 8.15M                       |   total crashes : 0 (0 unique)   |
| exec speed : 0.37/sec (zzzz...)           |   total tmouts : 11 (4 unique)   |
+- fuzzing strategy yields -----+- path geometry -----+
| bit flips : 256/287k, 16/286k, 10/286k       |   levels : 2   |
| byte flips : 0/35.9k, 5/35.9k, 3/35.9k       |   pending : 337   |
| arithmetics : 14/2.00M, 5/504k, 0/116k       |   pend fav : 41   |
| known ints : 3/207k, 7/1.14M, 12/1.39M       |   own finds : 337   |
| dictionary : 0/0, 0/0, 6/1.79M               |   imported : n/a   |
|   havoc : 0/204, 0/0                       |   stability : 96.64%   |
|   trim : 0.00%/2220, 0.00%                 +-----+
成功: 已终止 PID 为 3256 的进程。-----+ [cpu: 0%]

[-] PROGRAM ABORT : Cannot kill child process

```

通过 hook 的方式进行 fuzz

介绍

这种方式是之前从谷歌的一篇博客里面看到的。

<https://googleprojectzero.blogspot.com/2018/12/adventures-in-video-conferencing-part-1.html>

文章的大致思路是首先找出目标程序中负责数据解密的函数，然后利用 hook 的方式替换解密函数为一个 fuzz 的函数（功能其实就是随机填充数据）。之后当程序正常接收数据时，会首先调用解密函数先解密数据，然后在解密的数据进行处理。由于此时解密函数已经替换为了 fuzz 函数，所以我们相当于直接跳过了解密这个步骤，去 fuzz 解密后具体对数据进行处理的那部分逻辑，而往往会产生安全问题的恰恰就是真正处理数据的部分。这种 fuzz 方案的优势在于可以在不逆向加解密算法的基础上 fuzz 比较深层次的代码。

回到我们的目标程序。在处理新版本的皮肤文件时，会首先使用自定义的解密算法对数据进行解密。解密出来的数据是 zlib 压缩的，后面紧接着使用 zlib 解压缩。

```

buf_ = ptr;
new_size = size - 8;
v24 = ptr;
buf = ptr;
obj_ = sub_53BCF0();
if ( !decode_data_to_zlib(obj_, buf, &new_size, content_off_8, fsize_sub_8) )// 调用的参数为: obj,buf,file_content+8 file_size-8 .
// 会把文件内容 + 8 开始的数据解码, 解码后的数据为 zlib 格式, 保存在 buf。

goto LABEL_29;
size_4 = new_size - 4;
decoded_data_size = *buf_;

// 通过解码后的数据的开始4字节, 确定 zlib 解压的数据大小, 然后进行内存分配
// 这个 size 通过动态调试, 应该是动态解密出来的

v6 = decoded_data_size + 8;
v4 = get_mem_by_obj(&lpMem, 0, decoded_data_size + 8);// 整数溢出点: decoded_data_size + 8 , decoded_data_size应该可控, 通过调试时
v16 = header;
// 取到原始的头, Skin
*(v4 + 1) = HIWORD(header);
buf__ = v24;
*v4 = v16;
// 到现在为止, 在新分配的内存开始, 设置好了 Skin 的头部字段, 8 个字节 Skin + version
if ( zlib_decompress((v4 + 1), &decoded_data_size, buf__ + 4, size_4) )// 对刚 解码过的 zlib 数据, zlib 解压,
// buf__ 为 刚刚解码后的数据.
// 解压后的数据保存在 buffer + 8 出
// 解压后的长度保存在 decoded_data_size

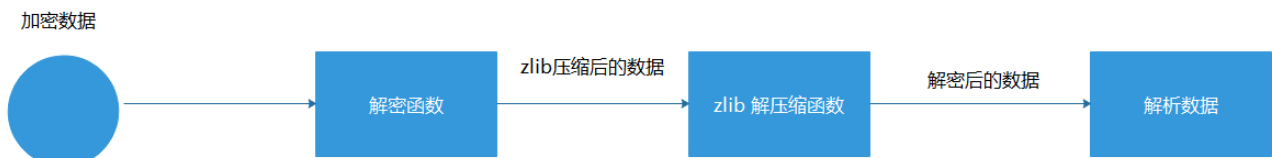
{
LABEL_29:
sub_482C20(&obj_);
}
00139DC9 handle_skinv3:71 (53A9C9)

```

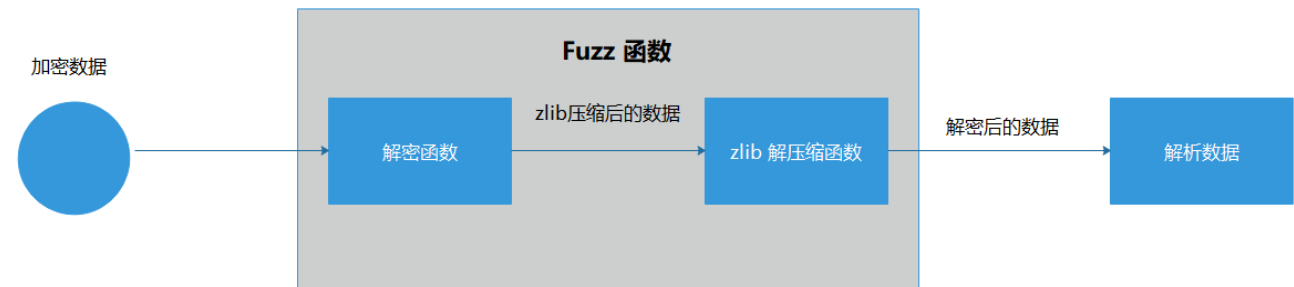
首先解密出 zlib 数据

然后解压缩 zlib 压缩后的数据

后面就开始处理进一步解析文件格式了。如下图所示



然后替换 解密函数 和 解压函数为 hook 函数就可以对 解析数据部分进行 fuzz 了。如下图所示



这里 hook 的方式采用 [mhook](#) 框架, 这个框架可以很方便的进行 inline hook.

```

#include "stdafx.h"
#include "tchar.h"
#include "mhook-lib/mhook.h"

```

```

typedef unsigned int(*decode_to_zlib)(char* out_buf,int* out_size, char* buf, unsigned int
size);
typedef unsigned int(*zlib_decompress)(char* out, int* decoded_data_size, char* buf,
unsigned int size);

```

```

void hexdump(FILE * stream, void const * data, unsigned int len)
{

```

```

unsigned int i;
unsigned int r, c;

if (!stream)
    return;
if (!data)
    return;

for (r = 0, i = 0; r < (len / 16 + (len % 16 != 0)); r++, i += 16)
{
    fprintf(stream, "%04X:  ", i); /* location of first byte in line */

    for (c = i; c < i + 8; c++) /* left half of hex dump */
        if (c < len)
            fprintf(stream, "%02X ", ((unsigned char const *)data)[c]);
        else
            fprintf(stream, "   "); /* pad if short line */

    fprintf(stream, " ");

    for (c = i + 8; c < i + 16; c++) /* right half of hex dump */
        if (c < len)
            fprintf(stream, "%02X ", ((unsigned char const *)data)[c]);
        else
            fprintf(stream, "   "); /* pad if short line */

    fprintf(stream, " ");

    for (c = i; c < i + 16; c++) /* ASCII dump */
        if (c < len)
            if (((unsigned char const *)data)[c] >= 32 &&
                ((unsigned char const *)data)[c] < 127)
                fprintf(stream, "%c", ((char const *)data)[c]);
            else
                fprintf(stream, "."); /* put this for non-printables */
        else
            fprintf(stream, " "); /* pad if short line */

    fprintf(stream, "\n");
}

fflush(stream);
}

void log_to_file(char *log) {
    FILE *pfile = fopen("c:\\log.txt", "a");
    fwrite(log, 1, strlen(log), pfile);
    fflush(pfile);
    fclose(pfile);
}

unsigned long long count = 0;
void fuzz(char* buf, int len) {

```



```

int q = rand() % 10;
if (q == 7) {
    int ind = rand() % len;
    buf[ind] = rand();
}
if (q == 5) {
    for (int i = 0; i < len; i++)
        buf[i] = rand();
}

char path[0x100] = { 0 };
snprintf(path, 0x100, "c:\\fuzz_%p.txt", count++);
FILE *pfile = fopen(path, "a");
hexdump(pfile, buf, len);
fflush(pfile);
fclose(pfile);
}

```

// 由于默认使用 stdcall，取参数时会从栈里面取，而目标函数的第一个参数为 this 指针，通过 ecx 取，所以不需要。

```

unsigned int hook_decode_to_zlib(char* out_buf, int* out_size, char* buf, unsigned int size)
{
    char log[0x100] = { 0 };
    snprintf(log, 0x100, "target:%p, buf:%p ,size:%p\n", out_buf, buf, size);
    //MessageBoxA(0, log, "hook_decode_to_zlib", MB_ICONEXCLAMATION);
    log_to_file(log);
    memcpy(out_buf, buf, size);
    return 1;
}

```

```

unsigned int hook_zlib_decompress(char* out, int* decoded_data_size, char* buf, unsigned int size)
{
    char log[0x100] = { 0 };

    unsigned int de_size = *decoded_data_size;
    snprintf(log, 0x100, "buf:%p ,de_size:%p, count: %p\n", buf, de_size, count);
    //MessageBoxA(0, log, "hook_zlib_decompress", MB_ICONEXCLAMATION);
    log_to_file(log);
    memcpy(out, buf, de_size);
    fuzz(out, de_size);
    return 1;
}

```

```

int __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    char* base = NULL;
    char* decode_to_zlib = NULL;
}

```

```

char* zlib_decompress = NULL;

switch (fdwReason)
{
case DLL_PROCESS_ATTACH://加载时候
    base = (char*)GetModuleHandle(_T("SGTool.exe"));
    decode_to_zlib = base + 0x239610;
    zlib_decompress = base + 0x4ffac0;

    /*
    // hook decode_to_zlib 函数
    if (Mhook_SetHook((PVOID*)&decode_to_zlib, hook_decode_to_zlib)) {
        char out[1024];
        snprintf(out, 1024, "base: %p, func:%p", base, decode_to_zlib);
        MessageBoxA(0, out, "inject", MB_ICONEXCLAMATION);
    }
    */

    // hook zlib_decompress 函数
    if (Mhook_SetHook((PVOID*)&zlib_decompress, hook_zlib_decompress)) {
        char out[1024];
        snprintf(out, 1024, "base: %p, func:%p", base, zlib_decompress);
        MessageBoxA(0, out, "inject", MB_ICONEXCLAMATION);
    }

    break;
default:
    break;
}
return TRUE;
return 0;
}

```

代码的逻辑是把 `zlib_decompress` (即 `zlib` 解压缩函数) 替换为 `hook_zlib_decompress` 函数。

```

unsigned int hook_zlib_decompress(char* out, int* decoded_data_size, char* buf, unsigned
int size)
{
    char log[0x100] = { 0 };

    unsigned int de_size = *decoded_data_size;
    snprintf(log, 0x100, "buf:%p ,de_size:%p, count: %p\n", buf, de_size, count);
    //MessageBoxA(0, log, "hook_zlib_decompress", MB_ICONEXCLAMATION);
    log_to_file(log);
    memcpy(out, buf, de_size);
    fuzz(out, de_size);
    return 1;
}

```

这个函数首先会对记录一些信息，然后把数据复制到 `out` 缓冲区，然后把 `out` 缓冲区的地址和大小传入 `fuzz` 函数，进行 `fuzz` 处理。

```
unsigned long long count = 0;
void fuzz(char* buf, int len) {
    srand(time(0));
    for (int i = 0; i < len; i++)
        buf[i] = rand();

    char path[0x100] = { 0 };
    snprintf(path, 0x100, "c:\\fuzz_%p.bin", count++);
    FILE *fp = fopen(path, "wb");
    fwrite(buf, 1, len, fp);
    fflush(fp);
    fclose(fp);
}
```

`fuzz` 函数就是往 `buf` 里面填随机数，然后保存样本到磁盘，以便后面进行复现。这里的 `fuzz` 函数写的比较简单，对数据的变异仅仅只是填充随机数。以后可以考虑借鉴 `af1` 的策略来提升 `fuzz` 的效率。编译 `hook` 代码会得到一个 `dll`。

Fuzz

首先使用

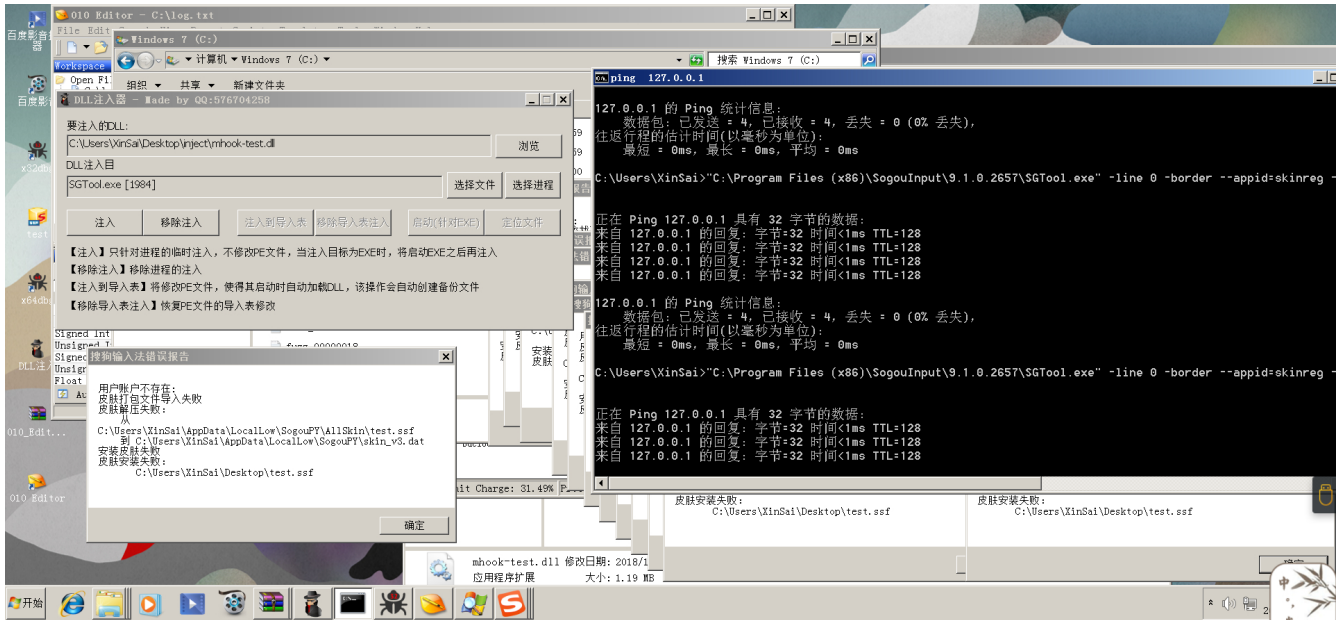
```
"C:\Program Files (x86)\SogouInput\9.1.0.2657\SGTool.exe" -daemon
```

创建一个守护进程，监听客户端的皮肤安装请求。然后使用 `dll` 注入工具，把生成的 `dll` 注入到进程中，然后用调试器附加上监控崩溃。

接下来就可以不断的发出安装请求，这样 **sgtool 守护进程** 就会不断调用解密函数，而此时我们已经 `hook` 了解密函数，则此时实际调用的是 `fuzz` 函数对数据进行变异，这样就对程序进行了 `fuzz`。

```
for /l %i in (1 1 1000) do "C:\Program Files (x86)\SogouInput\9.1.0.2657\SGTool.exe" -line
0 -border --appid=skinreg -install -c "C:\Users\XinSai\Desktop\test.ssf" -q -ef && ping
127.0.0.1
```

就是不断的安装皮肤，然后用 `ping` 命令来防止频率过高。



参考链接

<https://googleprojectzero.blogspot.com/2018/12/adventures-in-video-conferencing-part-1.html>

<https://sytheonp.github.io/2017/09/17/fuzzing-winapi.html>

<https://github.com/googleprojectzero/winapi>

<https://github.com/martona/mhook>