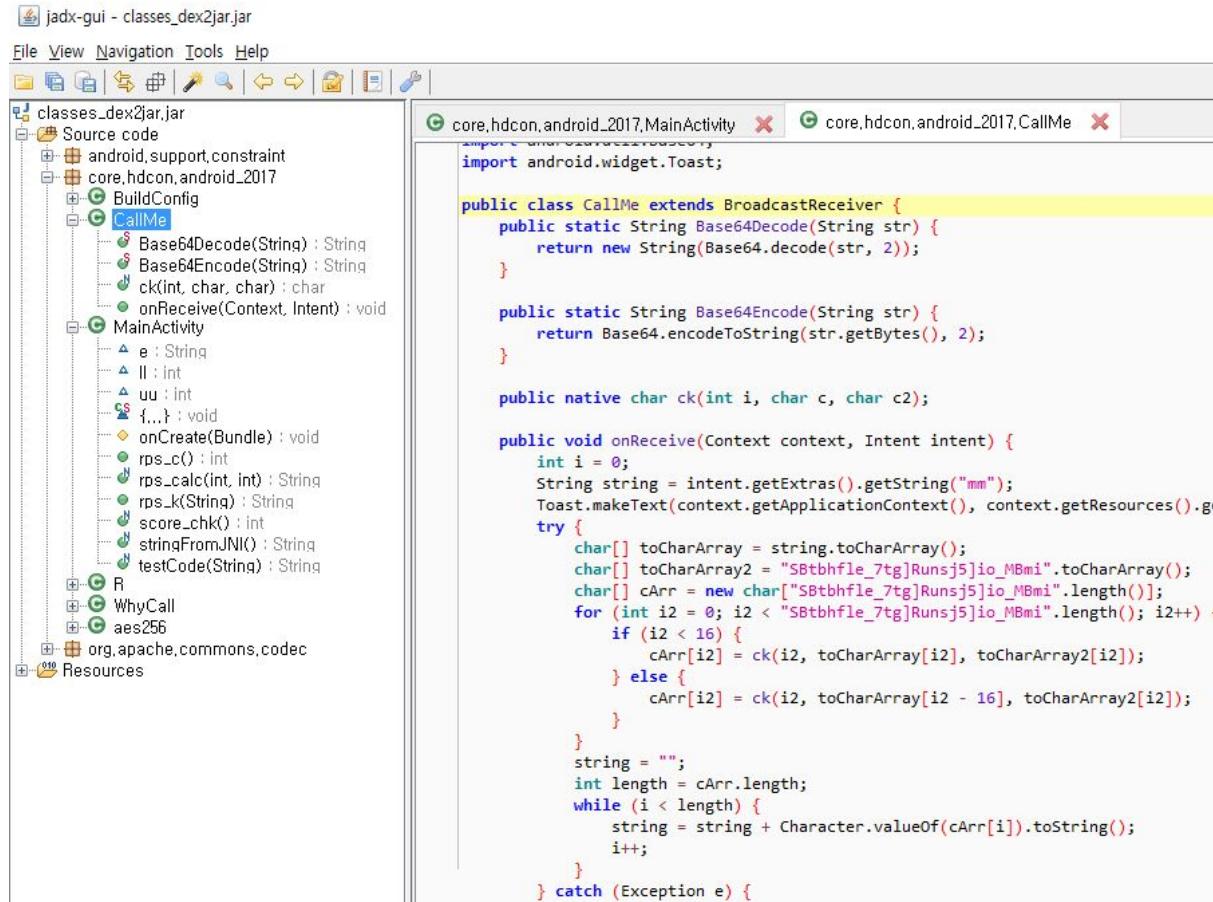


# HDCON 2017 Write up

- 한글영문또는특수문자금지라면길이제한은얼마일까자르짐ㅏㄹ아주세요 팀

## Feburary

가위바위보를 한다는 안드로이드 앱을 쳤다. 앱을 우선 까보니 아래와 같은 루틴이 있다. ck는 native 함수 콜이므로 앱에 들어있는 native library를 분석하도록 한다.



The screenshot shows the jadx-gui interface with the following details:

- File menu:** File, View, Navigation, Tools, Help.
- Toolbar:** Includes icons for file operations, search, and navigation.
- Left pane (File tree):** Shows the structure of the APK. It includes the main package 'core.hdcon.android\_2017' and its sub-classes: 'CallMe' and 'MainActivity'. 'CallMe' contains methods like 'Base64Decode', 'Base64Encode', and 'ck(int, char, char)'. 'MainActivity' contains methods like 'onCreate', 'onReceive', and 'score\_chk'.
- Right pane (Code editor):** Displays the Java code for the 'CallMe' class. It includes imports for 'android.widget.Toast' and 'java.util.Base64'. The code implements a 'BroadcastReceiver' and defines a static method 'Base64Decode' that returns a String. It also defines a static method 'Base64Encode' that returns a String. The most interesting part is the native method 'ck(int i, char c, char c2)' which is implemented in Java using JNI, calling a native function 'ck' from a library. The Java code then processes the result by creating a character array, applying the native function to it, and then concatenating the results into a string.

ck 함수는 들어온 값에 따라 간단한 값 처리를 한다.

```
signed __int16 __cdecl Java_core_hdcon_android_12017
{
    signed __int16 result; // ax@1

    LOBYTE(result) = a5;
    switch ( a3 % 6 )
    {
        case 0:
            LOBYTE(result) = ((a4 & 0xFFFF0u) >> 4) ^ a5;
            goto LABEL_8;
        case 1:
            LOBYTE(result) = ((a4 & 0xFFE0u) >> 5) ^ a5;
            goto LABEL_8;
        case 2:
            LOBYTE(result) = ((a4 & 0xFF80u) >> 7) ^ a5;
            goto LABEL_8;
        case 3:
            LOBYTE(result) = ((a4 & 0xFFC0u) >> 6) ^ a5;
            goto LABEL_8;
        case 4:
            goto LABEL_8;
        case 5:
            LOBYTE(result) = a5 ^ 0xF;
    LABEL_8:
        result = (char)result;
        break;
    default:
        result = 67;
        break;
    }
    return result;
}
```

자 이제 어떤 값이 input string으로 들어오는지 확인해보자. 이 역시 native 함수에서 값을 받아온다. 아래와 같다.

```

int __cdecl Java_core_hdcon_android_12017_MainActivity_rps_1calc(int a1, int a2)
{
    char *v4; // edi@1
    int v5; // ecx@1
    size_t v6; // esi@4
    _DWORD *v7; // eax@8
    char v9; // [esp+0h] [ebp-35Ch]@3
    char v10[20]; // [esp+8h] [ebp-354h]@1
    int v11; // [esp+1Ch] [ebp-340h]@1
    char v12; // [esp+4Bh] [ebp-311h]@1
    int v13; // [esp+14Ah] [ebp-212h]@1
    _int16 v14; // [esp+14Eh] [ebp-20Eh]@1
    int v15; // [esp+150h] [ebp-20Ch]@1
    _BYTE v16[3]; // [esp+249h] [ebp-113h]@1

    memcpy(v16, "You Win!!", 0xFFu);
    memset(&v15, 0, 0xF9u);
    v13 = 2002874948;
    v14 = 8481;
    v4 = &v12;
    memcpy(&v12, "You Lose..", 0xFFu);
    memcpy(&v11, "Detects abnormal behavior and reset to score 0", 0x2Fu);
    v10[16] = 0;
    *(_DWORD *)&v10[12] = 'W'='?=';
    *(_DWORD *)&v10[8] = '='='=';
    *(_DWORD *)&v10[4] = '^'='=';
    *(_DWORD *)v10 = 'S=TX';
    v5 = k;
    if ( k >= 17916 )
    {
        v4 = v10;
        if ( strlen(v10) )
        {
            v4 = v10;
            v9 = ae;
            v10[0] = ae ^ 0x58;
            if ( strlen(v10) >= 2 )
            {
                v6 = 1;
                v4 = v10;
                do
                {
                    v10[v6] ^= v9;
                    ++v6;
                }
                while ( v6 < strlen(v10) );
            }
        }
    }
}

```

최종적으로 코드를 정리하면 아래와 같은 코드가 나온다.

그런데 위에 input string을 xor해보면 HD-C-O-N-2-0-1-70이 나온다. 이것을 input string으로 넣고 계산을 한다면~? 역시 느낌이 젤 중요해~

```

import struct
a = [ 0x533D5458, 0x5E3D5F3D, 0x203D223D, 0x273D213D ]

def ck(a3, a4, a5):
    a4 = ord(a4)
    a5 = ord(a5)
    result = a5
    if a3 % 6 == 0:
        result = ((a4 & 0xFFFF0) >> 4) ^ a5
        pass
    elif a3 % 6 == 1:
        result = ((a4 & 0xFFE0) >> 5) ^ a5;
        pass
    elif a3 % 6 == 2:
        result = ((a4 & 0xFF80) >> 7) ^ a5;
        pass
    elif a3 % 6 == 3:
        result = ((a4 & 0xFFC0) >> 6) ^ a5;
        pass
    elif a3 % 6 == 4:
        pass
    elif a3 % 6 == 5:
        result = a5 ^ 0xf
        pass
    else:
        result = 0x43
        pass
    return chr(result)

a = struct.pack("<IIII", a[0], a[1], a[2], a[3]).decode()
ae = 0x10

s = ""
s += chr( 0x58 ^ ae)
for i in range(1, len(a)):
    s += chr(ord(a[i]) ^ ae)

print (a)
print (s)

print (repr(s))
s = s
t = "SBtbfhle_7tg]Runsj5]io_MBmi"

out = ""
for i in range(len(t)):
    if (i<16):
        out += ck(i, s[i], t[i])
    else:
        out += ck(i, s[i-16], t[i])

print (out)

```

June

인젝션된 코드를 찾아 시리얼 키를 획득하시오.

문제 설명부터 어썸합니다. 인젝션된 코드는 뭐고 시리얼 키는 뭔지 감도 잡히지 않습니다. 첨부파일을 다운받으려고 하는데 이름이 need\_strong\_mental.zip 입니다. 파일 제목부터 대놓고 멘탈을 뜯배기 깨 부수듯 부수겠다고 당당히 선언하는 모양새를 보니 닥쳐오는 재난 앞에 갈 곳을 잃은 기분입니다. 압축을 풀어보니 memory.img가 나옵니다. 대충 윈도우 메모리 덤프인 것 같습니다.

```
p1to ~/Downloads/hdcon $ vol.py -f memory.img imageinfo
Volatility Foundation Volatility Framework 2.6
INFO    : volatility.debug    : Determining profile based on KDBG search...
Suggested Profile(s) : Win7SP1x86_23418, Win7SP0x86, Win7SP1x86
                        AS Layer1 : IA32PagedMemoryPae (Kernel AS)
                        AS Layer2 : FileAddressSpace (/Users/hyeonseop/Downloads/hdcon/memory.img)
                        PAE type : PAE
                        DTB : 0x185000L
                        KDBG : 0x82d6ebe8L
Number of Processors : 1
Image Type (Service Pack) : 0
                        KPCR for CPU 0 : 0x82d6fc000L
                        KUSER_SHARED_DATA : 0xffffdf00000
Image date and time : 2017-09-15 11:53:39 UTC+0000
Image local date and time : 2017-09-15 20:53:39 +0900
p1to ~/Downloads/hdcon $
```

이것저것 해 봅니다. chrome.exe가 최근에 급작스럽게 서거하신 상황입니다만 별다른 흔적을 남기지는 않았습니다. 다른 프로세스들도 잘 알려진 것들 뿐 특별히 수상한 친구는 없습니다. 더 해볼 만한 게 있나 Volatility cheat sheet을 앞뒤로 넘겨보다가 눈에 확 띠는 커맨드 하나를 찾습니다. 역시 해킹은 툴빨입니다.

<b>ll:</b>	<b>-r/-regex=REGEX</b> Regex module name <b>-b/-base=BASE</b> Module base address	<b>Audit the II</b> idt (x86 on) gdt (x86 on)
<b>ck:</b>	<b>Injected Code</b>	<b>Audit drive</b> driverirp -r/-rege:
<b>gs</b>	Specify -o/--offset=OFFSET or -p/--pid=1,2,3	
	<b>Find and extract injected code blocks:</b> malfind -D/-dump-dir=PATH Dump findings here	<b>Display dev</b> devicetree
<b>ses:</b>	<b>Cross-reference DLLs with memory mapped files:</b> ldrmodules	<b>Print kerne</b> pooltracker -t/-tags -T/-tag

```

Process: explorer.exe Pid: 1940 Address: 0x31ad0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 7, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x31ad0000 55 8b ec 51 c7 45 fc 00 00 00 00 8b 45 08 0f be    U..Q.E.....E...
0x31ad0010 08 85 c9 74 14 8b 55 08 83 c2 01 89 55 08 8b 45    ...t..U.....U..E
0x31ad0020 fc 83 c0 01 89 45 fc eb e2 8b 45 fc 8b e5 5d c2    .....E....E...].
0x31ad0030 04 00 cc    ..... .

0x31ad0000 55          PUSH EBP
0x31ad0001 8bec        MOV EBP, ESP
0x31ad0003 51          PUSH ECX
0x31ad0004 c745fc0000000000 MOV DWORD [EBP-0x4], 0x0
0x31ad000b 8b4508      MOV EAX, [EBP+0x8]
0x31ad000e 0fbe08      MOVSX ECX, BYTE [EAX]
0x31ad0011 85c9        TEST ECX, ECX
0x31ad0013 7414        JZ 0x31ad0029
0x31ad0015 8b5508      MOV EDX, [EBP+0x8]
0x31ad0018 83c201      ADD EDX, 0x1
0x31ad001b 895508      MOV [EBP+0x8], EDX
0x31ad001e 8b45fc      MOV EAX, [EBP-0x4]
0x31ad0021 83c001      ADD EAX, 0x1
0x31ad0024 8945fc      MOV [EBP-0x4], EAX
0x31ad0027 ebe2        JMP 0x31ad000b
0x31ad0029 8b45fc      MOV EAX, [EBP-0x4]
0x31ad002c 8be5        MOV ESP, EBP
0x31ad002e 5d          POP EBP
0x31ad002f c20400      RET 0x4
0x31ad0032 cc          INT 3
0x31ad0033 cc          INT 3
0x31ad0034 cc          INT 3
0x31ad0035 cc          INT 3
0x31ad0036 cc          INT 3
0x31ad0037 cc          INT 3
0x31ad0038 cc          INT 3

```

인젝션한 코드를 찾아 준다는게 코드를 인젝션하려면 페이지를 새로 만들어서 거기다 어셈을 적어 넣을 것이기 때문에 read/write/execute 권한이 전부 있고 가장 앞에 어셈 비슷한게 들어있는 페이지를 찾아 준다는 것 같습니다. 쭉쭉 내려보는데 explorer.exe에 수상한 친구들이 잔뜩 나옵니다. 딱 봐도 페이지 새로 할당해서 함수들을 적어놓은 것 같습니다. explorer.exe의 메모리를 덤프하고 저 페이지들을 찾아서 거기 있는 함수들을 분석하니 대충 다음과 같은 느낌의 코드가 보였습니다.

```

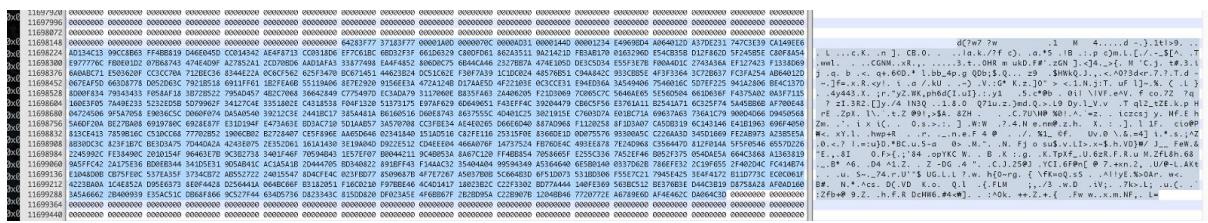
2     this[0] LoadLibrary(kernel32)
3     v4=kernel32
4     v8 = 0
5     this[4]
6     v10 = sleep
7     v18 = getmodulefilename
8
9     while (true) {
10         sleep(3000);
11         getmodulefilename(NULL, v19c, 260);
12         this_18(v19c, v98, this); // cut name
13         // correct v98 is "korea\0".
14         r = this_8(v98); // hash
15         if (r != 3ECF2406)
16             continue;
17
18         if (v8 == 0) {
19             v8 = this_0c(v98, this); // calc
20             if (v8 == 0)
21                 continue;
22         }
23
24         v14 = virtualprotect;
25         virtualprotect(&this_1C, 0x80, 0x40, &v0c); //rwx, size=0x80
26         this_1C(this);
27     }
28 // 1337000
29

```

(일부입니다)

분석을 통해 짜 맞춰 보자면, 소문자 5글자로 예상되는 현재 파일의 이름을 가져와서 해싱 비슷한 처리를 한 뒤, 그 결과를 0x3ecf2406과 비교합니다. 같다면 넘어가 파일 이름을 이용해 메모리의 특정 구역에 복호화를 수행하고 해당 위치로 EIP를 옮겨 코드를 실행합니다. 따라서 저 해시를 맞출 수 있으면 메모리에 수행하는 복호화를 따라해 볼 수 있고, 최종적으로 실행되는 코드가 뭔지도 알 수 있을 것입니다. 소문자 5글자는  $26^5$ , 약 천만 가지의 경우의 수가 있고 이를 모두 시도해 본 결과 korea가 나왔습니다.

분석한 코드가 특정 객체를 인자로 전달해가면서 실행되고, 마지막으로 실행되는 코드도 해당 객체에 들어있어 해당 객체를 메모리에서 찾아보았습니다. 여기 있네요



이제 아까 분석한 것과 같은 복호화를 수행한 뒤 실행하는 코드가 뭔지 들여다보면 되겠습니다.

```

1  |s = '64283F77 37183F77 00001A0D 0000070C 0000AD31 0000144D 00001234 E4969BD4 /'
2  |s = map(lambda x: int(x.decode('hex')[:-1].encode('hex'), 16), s.split())
3  |print len(s)
4
5  |a1 = map(ord, 'korea')
6  |v = [0]*20
7  |v[3] = a1[1] ^ a1[0];
8  |v[4] = a1[2] ^ a1[1];
9  |v[5] = a1[3] & a1[2];
10 |v[6] = a1[3] - a1[4];
11 |v[7] = a1[0] * a1[4] + a1[2];
12 |v[8] = a1[3] - a1[0] / a1[1];
13 |v[9] = a1[3] * a1[2] % a1[0];
14 |v[10] = a1[1] ^ a1[4] * a1[4];
15 |v = v[3:11]
16 |v = map(lambda x: x&0xff, v)
17
18 |corpus = s[39:]
19 |out = s[7:7+0x20]
20 |for j in range(0x20):
21 |    out[j] ^= s[39 + v[j % 8]]
22
23 |import struct
24 |code = ''
25 |for j in range(0x20):
26 |    code += struct.pack('<I', out[j])
27 |print code
28
29
30

```

실행해보니 뭔가 어셈같은게 나오는데, 그것보다 일단 딱 봐도 UUID같은 스트링이 들어있습니다. 어셈 분석따위 할 틈 없이 그대로 인증해봅니다. 키는 F7FE577B-1EC0-463F-90C4-614A799817D2 입니다. 파일 제목은 호기롭게 멘탈 박살낸다고 겁을 줬지만 생각보다는 정상적인 문제였던 것 같습니다. 실종된 개연성은 본선에서 찾아뵙는것으로 알고 있겠습니다.

July

주어진 메일함 파일에서 비정상적인 첨부파일을 포함한 메일을 찾고, 첨부 파일을 분석하여 플래그를 획득하시오.

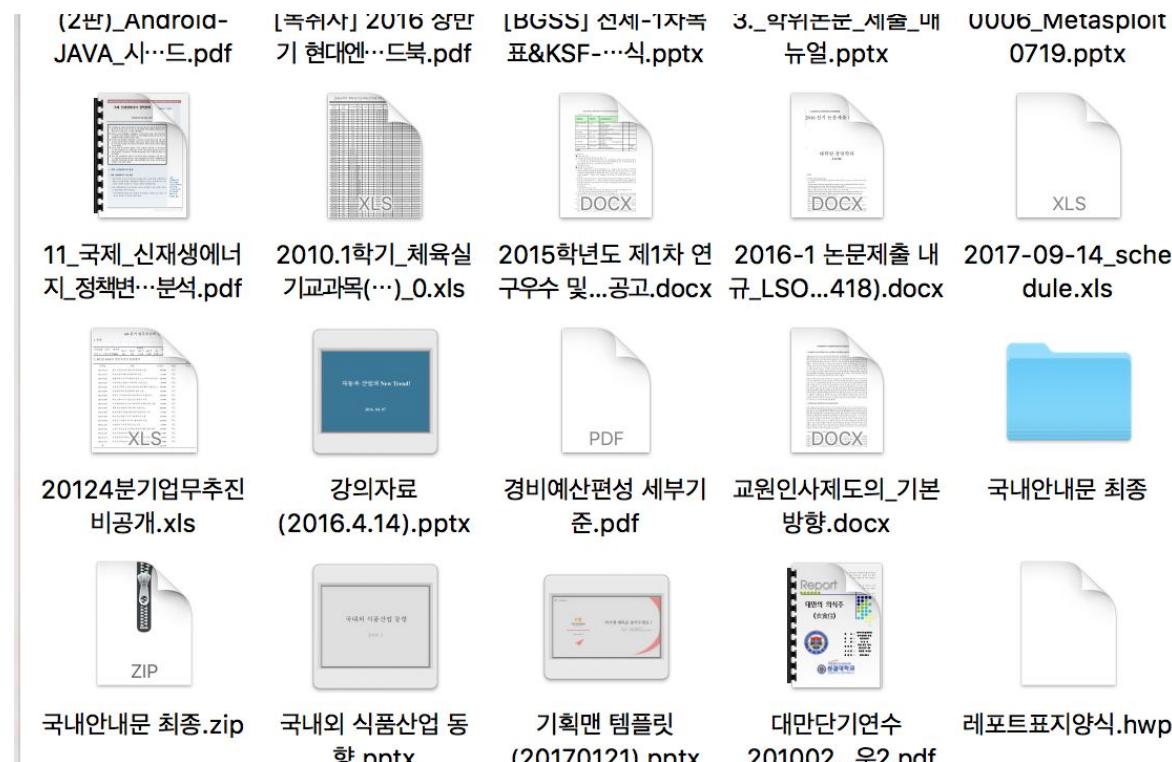
메일함이 도대체 뭔가 하고 받아봤더니 PST 파일이 들어있습니다. 뭔지는 잘 모르겠는데 대충 받은 메일들이 뭉테기로 들어 있는 무언가인 것 같습니다. 구글이 알려준 적당한 툴을 이용해 풀어해쳐봅니다.

```
demantos@daum.net ~ % Items done, 0 items skipped.
p1to ~/Downloads/hdcon $ file demantos@daum.net.pst
demantos@daum.net.pst: Microsoft Outlook email folder (>=2003)
p1to ~/Downloads/hdcon $ readpst -o pst demantos@daum.net.pst
Opening PST file and indexes...
Processing Folder "지운 편지함" 0 items done, 0 items skipped.
Processing Folder "지운 편지함" 0 items done, 1 items skipped.
Processing Folder "받은 편지함" 139 items done, 3 items skipped.
Processing Folder "보낸 편지함" 0 items done, 0 items skipped.
Processing Folder "보낸 편지함" 0 items done, 1 items skipped.
Processing Folder "보낸 편지함" 0 items done, 1 items skipped.
Processing Folder "정크 메일" 0 items done, 0 items skipped.
Processing Folder "대화이동작 설정" 0 items done, 0 items skipped.
Processing Folder "빠른 단계 설정" 0 items done, 0 items skipped.
"demantos@daum.net" 7 items done, 0 items skipped.
p1to ~/Downloads/hdcon $
```

스킵되는 친구들이 생기는데 왜 생기는지 모르겠습니다. 어떻게 해도 계속 스킵을 하는데 일단은 그냥 넘어가봅니다. 결과물을 보니 드디어 파일로 뜯어볼 수 있는 형식의 파일이 나왔습니다. 귀도아저씨는 대충 서쪽에 계실텐니 그 쪽으로 절 한번 드리고 첨부파일들을 추출해 보겠습니다.

```
1 # coding=utf-8
2 from __future__ import unicode_literals
3
4 import base64
5 import mailbox
6
7 mbox = mailbox.mbox('받은 편지함.mbox')
8
9 for title in mbox.values():
10     for msg in title.walk():
11         content_type = msg.get_content_type()
12         if content_type.startswith('multipart'):
13             continue
14         payload = msg.get_payload()
15         if 'Content-Transfer-Encoding' in msg.keys():
16             encoding = msg.get('Content-Transfer-Encoding')
17             if encoding == 'base64':
18                 payload = base64.b64decode(payload)
19             try:
20                 if content_type.startswith('text'):
21                     with open(subject + '.html', 'wb') as f:
22                         f.write(payload)
23                     subject += '_'
24                 else:
25                     with open(msg.get_filename(), 'wb') as f:
26                         f.write(payload)
27             except Exception as e:
28                 print(e)
```

뭔가가 잔뜩 나옵니다. 좋은(?) 파일도 가끔 보이고, 어떻게 봐도 민감한 개인정보로 보여서 이게 이렇게 불특정 다수의 나쁜사람들한테 노출되어도 출제진이 법적인 책임을 질 필요가 없는가에 대한 고민이 끝없이 떠오르는 자료들도 많습니다. 백개 좀 넘었던 것 같은데, 쪽쪽 내리다보면 겁나 수상한 pdf 파일이 하나 있습니다.



뭐가 수상하냐면 제목은 그럴듯한데 썸네일이 휩니다. 농담 아니고 진짜 이렇게 찾았습니다. 한글뷰어따위는 깔려있지않으므로 hwp파일이 수상했으면 영원히 풀지 못했을 문제입니다. pdf를 열어보니 swf파일이 포함되어있는 것 같은데, 파일 이름인 fancyball.swf를 검색해보니 무려 플래시의 취약점을 이용했던 알려진 멀웨어라고 합니다. ㅎㄷㄷ

#### Browser attack. Analysis of the malicious Flash objects and PDF ...

[www.noburkum.ru/analytics/en-flash](http://www.noburkum.ru/analytics/en-flash) ▾ 이 페이지 번역하기

2012. 3. 17. - Now we know the principles of analysis of both PDF and SWF files. ... Decompilation of fancyball.swf program results with innocent script, ...

#### PDF에 플래시? 어도비 PDF / 플래시의 또 다른 취약점 - Naked Security

<https://nakedsecurity.sophos.com/ko/2009/07/.../flash-pdf-vulnerability-adobe-pdfflash...> ▾

2009. 7. 24. - PDF 파일은 두 개의 EXE 파일과 두 개의 SWF 파일이 포함되어 있습니다. ... 여기에서 우리는 두 개의 SWF 파일 (fancyBall.swf 및 oneoff.swf) 참조 ...

#### Flash in the PDF? Another vulnerability with Adobe PDF/Flash ...

<https://nakedsecurity.sophos.com/.../flash-pdf-vulnerability-adobe...> ▾ 이 페이지 번역하기

2009. 7. 24. - The first SWF, fancyBall.swf, is just a simple little flash with a ball that will crash and the will allow the second code in the second SWF file to run.

#### Adobe Flash Exploit embedded inside PDF file | Security News

[www.pctools.com/.../adobe-flash-exploit-embedded-inside-pdf-fil...](http://www.pctools.com/.../adobe-flash-exploit-embedded-inside-pdf-fil...) ▾ 이 페이지 번역하기

2011. 1. 6. - One of them, fancyball.swf, doesn't seem to do anything malicious, the other flash file save.swf

친절하게 써 주신 포스트에 감사하며 찬찬히 읽어보니, 어떻게 heap spray를 일으켜서 원하는 코드를 실행시키고, 해당 코드는 pdf 파일에 들어있는 파일을 xor로 디코드해서 실행하는 방식의 멀웨어였다고 합니다. swf 파일을 찬찬히 살펴보니 내용이 완전히 같지는 않지만 큰 틀에서 차이는 없어 보였습니다. 적어놓지는 않았지만 xor에 사용하는 키를 여기서 추출했습니다.

다시 pdf 파일로 돌아와서 멀웨어가 드랍해내는 파일을 추출해야 하는데 어디에 있는지 찾기가 귀찮습니다. 적당히 pdf 파일 전체를 xor해 보니 중간에 jiff 헤더가 보입니다. 이미지인 모양이네요. 이미지를 뽑아 보니 플래그 비슷한게 보이는데 이미지가 살짝 깨쳤는지 아래쪽 일부가 보이지 않습니다. 앞부분이 `{lt_is_not_ov3r_until_it_` 인 것으로 미루어 짐작해 뒤에 `is_ov3r}`를 마음 속에서 붙여보았고 인증 했더니 맞았습니다.



## August

악성코드에 털린 사람의 브라우저 캐시랑 Temp 폴더 등이 포함된 압축파일이 있었다. 여기저기를 뒤져보다가 User\_Temp 폴더에 yqoivwolhpiocaoaiwc391634606.exe라는 수상한 프로그램이 있는 것을 발견했고, 분석해보니 악성코드 파일이었다. 레지스트리에 AppEvents에 JavaScript랑 Windows Host라는 값을 만드는데 JavaScript에는 Windows Host에 있는 값을 이용해 bin.exe를 만들어 실행하는 듯 하다. Windows Host에 들어가는 바이너리는 리소스에 들어있어서 뽑아내서 분석할 수 있었다. 사실 User\_Temp에도 남아있는 듯 하다. Diaspora라는 SNS에서 명령을 받아와서 뭔가 하는 듯한데, 플래그 형식에 적혀있는 로그인 아이디가 Diaspora 계정인 듯하여 로그인 부분을 분석해봤다. 간단한 암호화로 되어있는데, loudpeople.com에 markone이라는 계정에 markone2@이라는 패스워드를 쓴다. 뒤에 기능은 플래그와는 연관이 없는 듯하여 자세한 분석은 하지 않았다. 플래그 형식을 보면 감염 원인이 된 파일과 악성코드를 받아온 url이 필요한데, 악성코드는 yqoivwolhpiocaoaiwc391634606를 받아온 것을 의미하는 듯하여 해당 파일이 생긴 9월 14일 오후 5시 전후로 생긴 파일들을 위주로 분석했다.

### IECacheView.exe를 이용해서 IE에 남아있는 정보를 보니 아래와 같이

stM6O7LMDJ		http://convhkmp.taboola.com/st?cijs=convusm...	2017-09-14 오후 4:39:17
template[1].hta	application/hta	http://192.168.11.109/template.doc	2017-09-14 오후 5:00:20
hckmd[1].exe	application/x-msd...	http://www.marvel.com/media/ironman/hckmd...	2017-09-14 오후 5:00:34
owa.MailMessa...	application/javasc...	https://ow1.res.office365.com/owamail/201709...	2017-09-14 오후 5:09:23
owa.ReplyOrRen...	application/javasc...	https://ow1.res.office365.com/owamail/201709...	2017-09-14 오후 5:09:24

192.168.11.109에서 template.hta를 받고, www.marvel.com에서 hckmd.exe를 연달아 받은 흔적이 있었다. hckmd가 악성코드 파일이고 template.hta가 감염 원인인 줄 알았는데, 인증해보니 아니었고, 해당 시간 관련 파일들을 더 찾아보니

 ~WRS{5AA681A2-3A34-4F94-9832-657C8944BD44}.tmp	C:\사용자\hahah\바탕 화면\hdcon\august\pen...	유형: TMP 파일	수정한 날짜: 2017-09-14 오후 5:00 크기: 1.72KB
 ~WRS{5FC5B457-E6A8-4AC5-B51A-15EBB548CBC5}.tmp	C:\사용자\hahah\바탕 화면\hdcon\august\pen...	유형: TMP 파일	수정한 날짜: 2017-09-14 오후 5:00 크기: 1.00KB
 ~WRF{417AC73E-1278-47C8-A9AF-B516C4A17FE1}.tmp	C:\사용자\hahah\바탕 화면\hdcon\august\pen...	유형: TMP 파일	수정한 날짜: 2017-09-14 오후 5:00 크기: 16.0KB
 65FD2084.wmf		유형: WMF File	크기: 86바이트
 EB970CCF.wmf		유형: WMF File	크기: 86바이트
 {D0B66A99-C3D6-4541-AC08-8ECBF663F7A3} - OProcSessId.dat	C:\사용자\hahah\바탕 화면\hdcon\august\pen...	유형: DAT - MPEG 동영상 파일	수정한 날짜: 2017-09-14 오후 5:00 크기: 0바이트
 report	C:\사용자\hahah\바탕 화면\hdcon\august\pen...	유형: 바로 가기	수정한 날짜: 2017-09-14 오후 5:00 크기: 3.37KB
 report.rtf	C:\사용자\hahah\바탕 화면\hdcon\august\pen...	유형: 바로 가기	수정한 날짜: 2017-09-14 오후 5:00 크기: 4.70KB

report.rtf라는 파일이 있었고, ~WRF 파일이 report.rtf를 열었을 때 생긴 임시 파일로 추정된다. WRF 파일을 보면 template.doc의 url이 있어서 report.rtf가 감염 원인이고, 안에 있는 링크를 눌렀다가 감염된 스토리인 듯 하다.

그래서 답은

report.rtf [http://www.marvel.com/media/ironman/hckmd.exe\\_markone](http://www.marvel.com/media/ironman/hckmd.exe_markone)  
이다

## September

wanna\_smile이라는 랜섬웨어 파일과 털렸을 때 뜯 것 같은 메모리 덤프, 그리고 decryptme.docx.smile이라는 파일이 주어졌다.

우선 wanna smile부터 분석을 해보았다.

우선 RSA 2048 키를 생성하고, 비밀키는 하드코딩된 공개키를 이용해 암호화하여 “C:\wanna\_smile\00000000.eky”에 저장하고 공개키는 “C:\wanna\_smile\00000000.pky”에 저장한다.

그리고 “C:\”에서 특정 확장자 (문서, 소스코드 파일 등)를 가진 파일들을 찾아 암호화를 한다. 각 파일별로 AES키를 생성하여 암호화를 하는데, 이 AES키를 만들어둔 공개키를 이용해 암호화하여 파일의 앞에 붙여둔다.

암호화된 파일은 “WANASML!”이라는 매직값과, 4byte AES키 길이, 암호화된 AES키값 (256바이트), 4byte NULL, 4byte 파일길이, 그리고 파일 내용으로 이루어진다.

메모리 덤프를 확인하던 중 아래와 같이 RSA PRIVATE KEY가 아직 메모리에 남아있는 것을 발견했다.

11 02 67 D4 03 08 2D 2D 2D 2D 2D 2D  
1B 42 42 45 65 05 48 4D 43 43 43 43  
BEGIN ~~END~~ BEGIN KEY ..... MISC. ~~END~~

4D 49 49 45 6F 77 49 42 41 41 4B  
13 78 32 53 6A 61 68 44 31 4F 51  
1A 33 64 33 37 2B 75 30 4B 53 4A  
1A 31 42 34 42 5A 63 37 4D 41 5A  
14 73 36 45 4C 31 48 68 35 33 44  
1F 56 43 71 0A 33 31 53 6C 63 45  
14 67 78 35 61 43 39 4C 69 4B 6B  
18 73 68 58 33 79 55 54 65 71 51  
17 6E 45 55 70 58 63 2F 6C 4D 76  
13 79 75 4D 7A 39 53 32 6C 62 72  
1E 32 4E 6C 70 4D 72 35 6F 0A 43  
1A 44 48 73 6F 44 63 56 36 51 4F  
1A 59 36 61 45 4F 41 61 43 34 69  
1B 68 61 68 62 34 6B 56 55 64 62  
15 77 4E 32 34 67 52 34 51 4E 6D  
1F 79 46 6B 42 2F 32 45 59 44 38  
19 49 4E 7A 7A 5A 35 6A 59 76 6C  
1D 35 49 0A 33 54 37 42 55 38 64  
13 6F 56 37 67 62 75 77 2F 32 59  
12 67 51 44 35 5A 39 5A 62 44 33  
14 55 70 57 4E 42 2B 72 6E 45  
16 56 42 37 50 4E 44 35 41 69 6C  
18 4C 76 4F 7A 70 45 6F 0A 50 77  
16 58 4A 38 63 2B 30 76 55 6C 77  
BEGIN RSA PRIVATE KEY-----  
M11Eow1BAAKC  
CAQEAxCN92kGbxUYDzB2cDCTz8F3x2SjahD10Q  
41gR0UshN0DkUr1.8FKg3YVTJtTj3d37+u0KSJ  
wNekOJ1NtEptFXnGVAnEUzTkxOmj1B4BZc7MAZ  
9S1R.+1kXXj09yHyJ02F+sJzAV5ds6EL1Hh53D  
3W5GYzFy5tQH1UyUwa0YUZwLr1y/VGq.31S1cE  
9S1TLL10tYYpeZBEVfkqB0cvQNDWtg5ca5C91Lk  
OsxGboQvG5q8w53bKkly.67FUb7HshX3yUTEqQ  
5hK1wYeTUg3aaUdho3ep20SNb5FgnEuPxc/1Mv  
WAUtlasSw1.ZsXV9gmdQvHTvtW8PSyuMz9S2lbr  
sjSd31VMoQ1BAwKCAQEAgzJT5tZn2N1pMr5o.C  
BiiOpV2kMXnAt0Jg1uVg214dd+CjDHsoCvQ60  
Mxi3tPpP9UfNcmGgI/COGYzzY.bzY6aEOAc4i  
NDL4ZtOBQa7ondVm0d3hUZC6P6tN+hahb4kvUdb  
3Vj7pImtdoFBRP.X6PQu7MuH7zgEwN24gR4QNm  
gHhJ1gZUpkPkqwtEiTqzLDoYfkb/2EYD8  
KM5.JcZhYOHs7LJqDpAnkEmKm11INzzZ5jYv1  
/bBg8xB3UNsi0Wjh/9x1ly+J4MT51.3T7B8Ud  
dDQ7thnOb618MT934hEPYX2E713v07gbuw/2Y  
LwXs4yGtTyskNHHRf03.oVd7gwKBgQD5Z9Bd3  
Po9XXayKM28Se1REU/3zwhuAx+UTUpWNB+/rnE  
xKaJ5fgz.+thpqglCeCMPNS7143FvB7PND5Ail  
BvDChL88Lz1x3qMqQqxFV8R3j5+ZHLvOzpEo.Pw  
Yv...Pc6P4TBRPGChowvPi.EF8i...EFSy...Bz+uVJw

이 키를 뽑고, 암호화된 파일에서 AES 키 부분만 추출해서

```
openssl rsa -inkey priv.pem -in key -out key.out
```

로 키를 복호화 했고. (그런데 잘못짜서 키 뒤에 스택값이 붙어 나온듯하다..)

## AES 키가

“DEEC90A9B47BB62440CA32BE556EB7DE556FA41F71F60C512B7CCBD5706B9FA6”

라는 것을 알 수 있었다.

암호화된 파일 뒷부분에 파일 내용만 뽑아서

```
openssl aes-256-cbc -d -K
```

DEFC90A9B47BB62440CA32BE556FB7DE556FA41F71F60C512B7C9BD5706B9FA6 -iv

로 볼호화 했다. iv는 0으로 되어있었다.

뭔가 잘못 했는지 파일이 살짝 깨지기 했는데, 착한 MS Word가 알아서 복구해서 열어졌다.

답은 TIMEISGOLD!

October

DNS 패킷의 request query와 response data를 파싱해서 base64 decode를 하면 custom HTTP protocol 이 나옵니다..

앞의 7바이트가 헤더이고 나머지가 HTTP 데이터인데, 헤더에는 ID 빙스모리 한 것이 있어서 이걸 가지고 response들을 룩으면 됩니다.

패킷을 잘 살펴보면 RAR 파일을 받아오는데... 암호가 걸려있습니다.. 암호는 6글자 lower case 라고 써있었으니 CUDA로 브포를 하면 끝

비번은 batcar이였고, flag는 J0ker\_is\_no7\_R3dH00d!!



November

packet 파일 잔뜩 주고 decryption이 되는 트래픽이 있다고 문제가 나옴. SSL 패킷을 많이 줬으니 당연히 common prime 문제라고 생각함. 모든 패킷에서 n을 뽑고 각각의 n끼리 gcd를 구해서 private key를 찾고, 패킷을 복호화하면 되는 간단한 문제였다.

```
flist = glob.glob("*.pcap")

pubkeys = []

for fname in flist:
    print(fname)
    cap = pyshark.FileCapture(fname)
    s = ""
    for i in range(100):
        try:
            s = long_to_bytes(int("".join(cap[i].ssl.handshake_certificate.split(":"))),16))
            break
        except:
            pass
    if not s:
        print ("failed: ", fname)
        continue
    with open("a.der", "wb") as f:
        f.write(s)

    cmd1 = "openssl x509 -inform der -in a.der -out b.pem"
    cmd2 = "openssl x509 -pubkey -noout -in b.pem > c.pem"
    os.system(cmd1)
    os.system(cmd2)

    with open("c.pem", "r") as f:
        data = f.read()
        pubkeys.append(RSA.importKey(data))

flag = False
for j, n2 in enumerate(Ns):
    if j != i:
        n1 = pk.n
        val = gmpy2.gcd(n1, n2)
        if val != 1:
            print (flist[i], flist[j])
            print ("common factor")
            p = val
            q = n1 / p
            # Compute phi(n)
            phi = (p - 1) * (q - 1)

            # Compute modular inverse of e
            gcd, d, b = egcd(pk.e, phi)
            d = int(d) % phi
            d = int(d)
            keys.append(RSA.construct((pubkeys[i].n, pubkeys[i].e, d)))
            flag = True
        if flag:
            continue
```

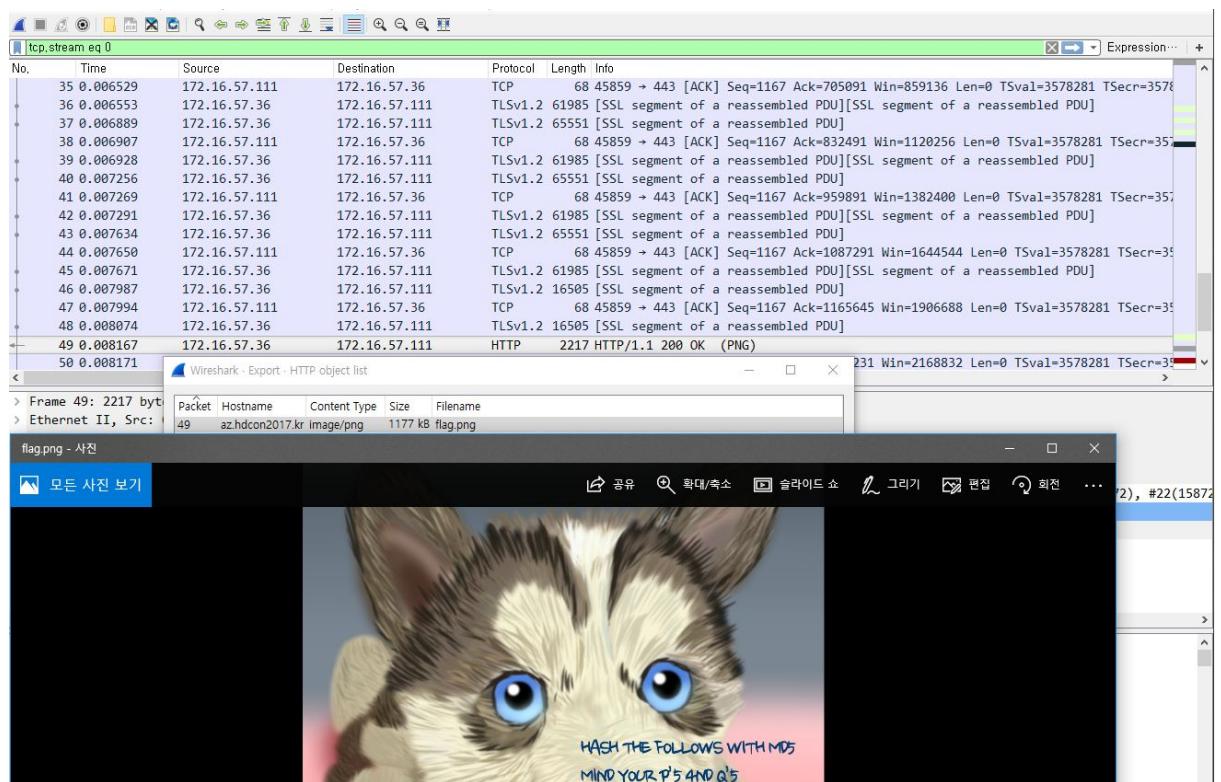
```

p = 0xfe67b5757bc0528325ee178c3a3cd4b0b675057084aeec5abaf42b5241efd606824c1b61660ff7cbdef7c72199bb3db86704776147ac75bfd6d141ae6cc72
adL
q = 0xf068ec5a7e2fe7987e109637957d9e4b5efa8661a61f2e665d5234fd4278282f29dc05e4d1f8d0bfd187b4e8b103fb3838b5f83222f850fdad7a48cce1c5
8bL

p = 0xfe67b5757bc0528325ee178c3a3cd4b0b675057084aeec5abaf42b5241efd606824c1b61660ff7cbdef7c72199bb3db86704776147ac75bfd6d141ae6cc72
adL
q = 0xf2284c95c41ab96ea178534e83dcf9c6947f325583f9268745730aae179a1ac681e42134a6757e559c88c36cdb5551754588a229ebe624808068ba432d4568
69L
n = p * q
e = 65537
phi = (p-1) * (q-1)
d = modinv(e, phi)
private_key = RSA.construct((long(n), long(e), long(d)))
with open("priv2.pem", "w") as f:
    f.write(private_key.exportKey())

```

이렇게 뽑은 private key를 가지고 wireshark의 ssl 패킷을 복호화한다.



참 쉽죠~?