Writeup: AlexCTF 2017

Reverse Engineering

Feb 9, 2017

Writeups

Reverse Engineering

My solutions to the five reverse engineering challenges from AlexCTF in February 2017.

Gifted

I'm not even gonna bother…

```
$ strings ./gifted
```

AlexCTF{Y0u_h4v3_45t0n15h1ng_futur3_1n_r3v3r5ing}

C++ is awesome

I started by throwing the binary into binary ninja to have a poke around. Binary Ninja didn't detect any of the subroutines automatically, so I scrolled through the `.text` section and manually identified sections as subroutines with `Shift-P`.

Reading through the control flow graph gave me a decent understanding of its function, but nothing beats stepping through the binary in GDB, so I loaded it on with a dummy flag and stepped through. After a few minutes, I identified that the instruction at `0x400c75` was comparing each character from my input string to another character, exiting the program if the check failed. After setting a breakpoint and looping past it a few times, I could see that the mystery characters were forming the flag. With this known, I could use r2pipe to run the binary up to that address, take the character of the flag, and add it to a buffer, then restart the binary with the buffer as the input, allowing execution to progress to get the next character./

```
import r2pipe

r2 = r2pipe.open("./re2")
r2.cmd("doo AB")
r2.cmd("db 0x400c75")
r2.cmd("dc")

flag = ""
```

```
while not flag.endswith('}'):
    char = r2.cmd("dr al")
    flag += chr(int("0x" + char[8:] ,16))
    r2.cmd("doo " + flag + "X")

    for i in range(0, len(flag)+1):
        r2.cmd("dc")


print(flag)
```

ALEXCTF{W3_L0v3_C_W1th_CL45535}

Catalyst system

Again starting with binary ninja, I manually went through the `.text` section and identified sections as subroutines with `Shift-P`.

When I first ran the binary, it took forever to load. I noticed that the reason for this was a loop at the beginning of `main()`. I bypassed it by using Binary Ninja to invert the branch conditions and save a patched copy of the binary.

```
00400ea5   837dfc1d        cmp    dword [rbp-0x4 {var_c}], 0x1d
00400ea9   7ebc            jle    0x400e67
```

↓

```
00400ea5   837dfc1d        cmp    dword [rbp-0x4 {var_c}], 0x1d
00400ea9   7fbc            jg     0x400e67
```

```
00400f5e   837df81d        cmp    dword [rbp-0x8 {var_10}], 0x1d
00400f62   7ec2            jle    0x400f26
```

↓

```
00400f5e   837df81d        cmp    dword [rbp-0x8 {var_10}], 0x1d
00400f62   7fc2            jg     0x400f26
```

Continuing reading through the disassembled binary, I found the first input validation subroutine at `0x400c9a`. It calculates the length of the username string and passes it to the subroutine at `0x400c41` via the `EDI` register. This subroutine performs four checks against the length of the username:

1. `(len >> 2) << 2 == len`
2. `(len >> 4) << 2 != (len >> 2)`
3. `len >> 3 != 0`
4. `len >> 4 == 0`

```
sub_400c41:
00400c41  55                push    rbp
00400c42  4889e5            mov     rbp, rsp {var_8}
00400c45  4883ec20          sub     rsp, 0x20
00400c49  897dec            mov     dword [rbp-0x14 {strlen}], edi
00400c4c  8b45ec            mov     eax, dword [rbp-0x14 {strlen}]
00400c4f  c1f802            sar     eax, 0x2
00400c52  8945fc            mov     dword [rbp-0x4 {strlen >> 2}], eax
00400c55  8b45fc            mov     eax, dword [rbp-0x4 {strlen >> 2}]
00400c58  c1e002            shl     eax, 0x2
00400c5b  3b45ec            cmp     eax, dword [rbp-0x14 {strlen}]
00400c5e  7523              jne     0x400c83
```

```
00400c60  8b45fc            mov     eax, dword [rbp-0x4 {strlen >> 2}]
00400c63  c1f802            sar     eax, 0x2
00400c66  8945f8            mov     dword [rbp-0x8 {strlen >> 4}], eax
00400c69  8b45f8            mov     eax, dword [rbp-0x8 {strlen >> 4}]
00400c6c  c1e002            shl     eax, 0x2
00400c6f  3b45fc            cmp     eax, dword [rbp-0x4 {strlen >> 2}]
00400c72  740f              je      0x400c83
```

```
00400c74  8b45fc            mov     eax, dword [rbp-0x4 {strlen >> 2}]
00400c77  d1f8              sar     eax, 0x1
00400c79  85c0              test    eax, eax
00400c7b  7406              je      0x400c83
```

```
00400c7d  837df800          cmp     dword [rbp-0x8 {strlen >> 4}], 0x0
00400c81  7414              je      0x400c97
```

```
00400c97  90                nop
00400c98  c9                leave
00400c99  c3                retn
```

```
00400c83  bf69104000        mov     edi, 0x401069  {"invalid username or password"}
00400c88  e843faffff        call    puts
00400c8d  bf00000000        mov     edi, 0x0
00400c92  e8b9faffff        call    exit
{ Does not return }
```

This means that the valid lengths for the username are either 8 or 12 characters long.

The next input validation subroutine is located at `0x400cdd`. It divides the username into three sections consisting of four bytes each (this answers the question of whether the username is 8 or 12 characters long). For the sake of example, X will refer to the first four characters, Y to the next four, and Z to the last four. This subroutine performs three checks:

1. `X - Y + Z == 0x5c664b56`
2. `3 * (X + Z) + Y == 0x2e700c7b2`
3. `Y * Z == 0x32ac30689a6ad314`

We can use Z3 to resolve the valid string

```
          from z3 import *

username = ""
x = Real('x')
y = Real('y')
z = Real('z')

s = Solver()
s.add(x - y + z == 0x5c664b56, 3 * (x + z) + y == 0x2e700c7b2,
s.check()

for d in s.model():
        print(hex(int("%s" % (s.model()[d]))))
        username += hex(int("%s" % (s.model()[d])))[2:].decode

print("Username found: " + username)  # "catalyst_ceo"
print(s.model())
```

> The third validation subroutine at `0x40087f` merely checks to make sure the username only consists of lowercase letters and underscores. Seeing as we already have the correct username, it is not necessary to do anything with it.

The final step in solving this challenge is the subroutine at `0x400977`. It contains a flaw, whereby the RNG is seeded using the sum of X, Y and Z mentioned above. This means that the values of `rand()` can be predicted. We use this C code to do that:

```
          // Compiled with "gcc srand2.c -o srand2 -Wno-overfl
#include <stdio.h>
#include <stdlib.h>

int main()
{
   srand(0x61746163 + 0x7473796c + 0x6f65635f);  // X + Y + Z
   for (int i = 0; i <= 9; i++) {
```

```
        printf("%d: 0x%08X\n",i ,rand());
    }
}
```

We can then take those “random” numbers and save them in a python list for later use. After all this, the code begins to cascade down through 10 blocks of code, each ending with a `cmp eax, <constant>; je <next-block>;` where `eax` is the result of subtracting 4 bytes of the password from a “random” number. Rather than going through every block and copying the constant manually, I used r2pipe to do if for me. It sets a breakpoint at each `cmp` using a for loop (because each breakpoint was at a fixed offset of `0x2e` and I’m lazy), then starts executing. When it hits the *nth* breakpoint, it takes the constant at the current instruction and adds it to the *nth* predetermined random number, therefor reversing the subtraction. The result is added to a string buffer and the code is restarted using the updated buffer as the password.

My complete exploit payload is located here:

```
          from z3 import *

import struct
import subprocess
import csv
import r2pipe
import sys

username = ""
password = ""


x = Real('x')
y = Real('y')
z = Real('z')

s = Solver()
s.add(x - y + z == 0x5c664b56, 3 * (x + z) + y == 0x2e700c7b2,
s.check()

for d in s.model():
    username += hex(int("%s" % (s.model()[d])))[2:].decode("he

print("Username found: " + username)


fo = open("credentials.txt", "wb")
fo.write(username + "\n" + "AAAA" + "\n")
```

```
fo.close()

fo = open("catalyst.rr2", "wb")
fo.write("#!/usr/bin/rarun2" + "\n" + \
        "program=./catalyst-patched" + "\n" + \
        "stdin=./credentials.txt" + "\n" + \
        "stdout=" + "\n")
fo.close()


r2 = r2pipe.open("./catalyst-patched")
r2.cmd("e dbg.profile=catalyst.rr2")
r2.cmd("doo")

for i in range(0, 10):
    break_addr = 0x00400a80 + (0x0000002e * i)
    r2.cmd("db " + str(break_addr))


random_ints = [ 0x00684749,
                0x673CE537,
                0x7B4505E7,
                0x70A0B262,
                0x33D5253C,
                0x515A7675,
                0x596D7D5D,
                0x7CD29049,
                0x59E72DB6,
                0x4654600D ]

for i in range(0, 10):
    r2.cmd("dc")
    current_instruction = r2.cmdj("pdj 1")[0]
    password += struct.pack('I', (int(random_ints[i] + int(cur

    fo = open("credentials.txt", "wb")
    fo.write(username + "\n" + password + "\n")
    fo.close()

    r2.cmd("doo")

    for j in range(0, i + 1):
        r2.cmd("dc")

print("Password found: " + password)
```

```
catalyst = subprocess.Popen("cat credentials.txt | ./catalyst-
stdout, stderr = catalyst.communicate()
print(stdout)
```

*ALEXCTF{1_t41d_y0u_y0u_ar3**gr34t**reverser__s33}*

---

unVM me

After downloading the file, I used the `file` utility to determine what I was working with.

```
$ file unvm_me.pyc
unvm_me.pyc: python 2.7 byte-compiled
```

So I used `uncompyle6` to decompile it.

```
$ uncompyle6 unvm_me.pyc > unvm_me.py
```

Giving me the source code.

```
# uncompyle6 version 2.9.8
# Python bytecode 2.7 (62211)
# Decompiled from: Python 3.6.0 (default, Jan 16 2017, 12:12:5
# [GCC 6.3.1 20170109]
# Embedded file name: unvm_me.py
# Compiled at: 2016-12-21 08:44:01
import md5

md5s = [174282896860968005525213562254350376167,
        137092044126081477479435678296496849608,
        126300127609096051658061491018211963916,
        314989972419727999226545215739316729360,
        256525866025901597224592941642385934114,
        115141138810151571209618282728408211053,
        8705973470942652577929336993839061582,
        256697681645515528548061291580728800189,
        39818552652170274340851144295913091599,
        65313561977812018046200997898904313350,
        230909080238053318105407334248228870753,
        196125799557195268866757688147870815374,
        74874145132345503095307276614727915885]

print 'Can you turn me back to python ? ...'
flag = raw_input('well as you wish.. what is the flag: ')

if len(flag) > 69:
    print 'nice try'
```

```
    exit()

if len(flag) % 5 != 0:
    print 'nice try'
    exit()

for i in range(0, len(flag), 5):
    s = flag[i:i + 5]
    if int('0x' + md5.new(s).hexdigest(), 16) != md5s[i / 5]:
        print 'nice try'
        exit()

print 'Congratz now you have the flag'
```

As we can see, the length of the flag must be a multiple of 5, and no longer that 68 characters. The verification loop takes the first five characters of the input, creates an MD5 hash, and compares it to the first hash in the md5s list. It then takes the next five characters of the input, creates another MD5 hash, and compares it to the second hash, and so on.

This seems like a good time to spin up a little brute forcing script:

```
          import md5
import itertools

md5s = [174282896860968005525213562254350376167,
        137092044126081477479435678296496849608,
        126300127609096051658061491018211963916,
        314989972419727999226545215739316729360,
        256525866025901597224592941642385934114,
        115141138810151571209618282728408211053,
        8705973470942652577929336993839061582,
        256697681645515528548061291580728800189,
        39818552652170274340851144295913091599,
        65313561977812018046200997898904313350,
        230909080238053318105407334248228870753,
        196125799557195268866757688147870815374,
        74874145132345503095307276614727915885]

chars = "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM1
flag = ""

for i in range(0, len(md5s)):
    for string in itertools.imap(''.join, itertools.product(''
        print "str: " + string + "\tflag: " + flag
        if int('0x' + md5.new(string).hexdigest(), 16) == md5s
```

```
            flag += string
            break

print flag
```

ALEXCTF{dv5d4s2vj8nk43s8d8l6m1n5l67ds9v41n52nv37j481h3d28n4b6v3k}

Packed Movement

Before I even looked at the file, the challenge title was a giveaway that the binary was probably packed. I verified that using `hexdump -C move | head` , then unpacked it with `upx -d move` . I then opened the unpacked binary in binary ninja and proceeded to shit my pants. I hadn't reverse engineered any obfuscated code before, and didn't really know where to begin. So I tried basic string searching hoping to luck out. Running `strings` against the binary yielded over one hundred and thirty five thousand results, and grepping through that gave me nothing. I then tried grepping the `objdump` for ASCII codes of the flag format "ALEXCTF" (`0x41, 0x4c, 0x45, 0x58, 0x43, 0x54, 0x46`), and noticed a pattern:

```
          $ objdump -d move-unpacked -M intel | grep 0x41
 80493db:       c7 05 68 20 06 08 41    mov    DWORD PTR ds:0x
$ objdump -d move-unpacked -M intel | grep 0x4c
 8049dde:       c7 05 68 20 06 08 4c    mov    DWORD PTR ds:0x
$ objdump -d move-unpacked -M intel | grep 0x45
 804a7e1:       c7 05 68 20 06 08 45    mov    DWORD PTR ds:0x
$ objdump -d move-unpacked -M intel | grep 0x58
 804b1e4:       c7 05 68 20 06 08 58    mov    DWORD PTR ds:0x
```

At one point or another throughout the programs execution, individual characters of the flag are moved into `ds:0x8062068` . So I identified a common string across all of these instructions to be " `DWORD PTR ds:0x8062068,` ", and used that in a nifty one liner to get the flag.

```
          objdump -d move-unpacked -M intel | grep "DWORD PTR
```

ALEXCTF{M0Vfusc4t0r_w0rk5_l1ke_m4g1c}

Writeup: CySCA 2014