

HGAME-Week4-Mezone



Web

Re:Go

做完后，觉得比想象中的简单好多。。

刚开始做的时候，没发现给出了后台程序。。

前台要求填入一个code来获取flag。

SHOW ME THE CODE NOW!

009575

GET FLAG

















拿到后台程序，拖IDA，发现符号表[无]了。

突然想到，之前在300B围观王子围观li4n0学长逆向Golang的时候，展示过恢复Go的一些符号的插件。于是去网站搜索，搜到了一些IDAPython脚本，在使用过程中遇到了不少麻烦。

之前在网站上下载的IDA7.4 (Demo)版本，没有带IDAPython，所以最新的脚本无法使用。

回到稳定的IDA7.0，尝试十多个脚本，都报了不同的错，十分令人绝望。

拿出被抛弃很久的IDA7.2，打开时提示python错误，搜索教程修复。再尝试使用脚本，成功恢复了函数名！

-  runtime_main_func1
-  runtime_main_func2
-  main_Service_InitConfig
-  main_Service_InitDataBase
-  main_Service_GetFlag
-  main_main
-  main_Service_InitRouter
-  main_Service_AuthRequired
-  main_Service_Init
-  main_Service_NewToken
-  main_Service_Register
-  main_Service_Login
-  main_Service_UpdateProfile
-  main_Service_GetProfile
-  type_hash_main_DB
-  type_hash_main_Server

直接看GetFlag函数，找到可疑位置：

```
else
{
    v28 = aShal;
    *(_QWORD *)&v29 = 4LL;
    *((_QWORD *)&v29 + 1) = crypto_sha1_New_ptr;
    SEED = aX5jmtfgt4fvj34;
    v31 = 16LL;
    v32 = 6LL;
    *(_QWORD *)&v33 = &v28;
    *((_QWORD *)&v33 + 1) = 30LL;
    SEED_1 = &SEED;
    github_com_xlzd_gotp__TOTP_Now(a1, a2, v9); |
    v15 = v26[1];
    if ( &unk_9EF9A0 == (_UNKNOWN *)v15
        && (SEED_1 = *v26, v24 = v26[1], runtime_memequal(a1, a2, v14, (char)v15), (_BYTE)v26) )
    {
        v21 = *(const char **)(*_QWORD *)a7 + 80LL;
        runtime_convTstring(a1, a2, v19);
        runtime_makemap_small(a1);
        runtime_mapassign_faststr(a1, a2, (__int64)aError_0, (__int64)&Map_item_2);
        MEMORY[0] = &unk_A07EC0;
        if ( dword_1016B90 )
        {

```

去GitHub上面看了一下调用的函数GOTO。

GOTP - The Golang One-Time Password Library

[build-status](#) [MIT License](#)

GOTP is a Golang package for generating and verifying one-time passwords. It can be used to implement two-factor (2FA) or multi-factor (MFA) authentication methods in anywhere that requires users to log in.

Open MFA standards are defined in [RFC 4226](#) (HOTP: An HMAC-Based One-Time Password Algorithm) and in [RFC 6238](#) (TOTP: Time-Based One-Time Password Algorithm). GOTP implements server-side support for both of these standards.

GOTP was inspired by [PyOTP](#).

基于时间的一次性密码生成器。

看一下例子：

Time-based OTPs

```
totp := gotp.NewDefaultTOTP("4S62BZNFXXSZLCRO")
totp.Now() // current otp '123456'
totp.At(1524486261) // otp of timestamp 1524486261 '123456'

# OTP verified for a given timestamp
totp.Verify('492039', 1524486261) // true
totp.Verify('492039', 1520000000) // false

// generate a provisioning uri
totp.ProvisioningUri("demoAccountName", "issuerName")
// otpauth://totp/issuerName:demoAccountName?secret=4S62BZNFXXSZLCRO&issuer=issuerName
```

结合代码

```
SEED = aX5jmtfgt4fvj34; // X5JMTFGT4FVJ34GV
v31 = 16LL;
v32 = 6LL;
*(_QWORD *)&v33 = &v28;
*((_QWORD *)&v33 + 1) = 30LL;
SEED_1 = &SEED;
github_com_xlzd_gotp__TOTP_Now(a1, a2, v9);
```

猜测使用了基于当前时间的一次性密码。

似乎要写第一个Go程序了？

从网上下载Go的编译器，新建项目，

最近添加



GoLand 2019.3.2

导入上面的包，

```
package main
import (
    "fmt"
    "github.com/xlzd/gotp"
)
func main() {
    fmt.Println(a...: "Current OTP is", gotp.NewDefaultTOTP("X5JMTFGT4FVJ34GV").Now())
}
```

运行多次，发现这个密码的刷新频率超过了1秒。

于是运行，当密码变化时，抓紧填入网站，提交！

得到了flag? ! ? !

SHOW ME THE CODE NOW!

009575

GET FLAG

hgame{GO_Web_Application_With_RE_Thank_you_for_playing_it!}

那这个sha1做什么用的?? 迷惑。

```
v28 = aSha1;  
*(_QWORD *)&v29 = 4LL;  
*((_QWORD *)&v29 + 1) = crypto_sha1_New_ptr;
```

RE

Secret

下载,运行,输入即flag.

IDA,查看使用字符串的函数(main),发现程序向服务器索取了一段东西,填到内存中执行.于是修改二进制,将索取的东西填到程序里,重新IDA.

注意到,程序运行后,fork了一下,然后共同协作,完成对输入的加密,验证.

根据分析和调试,程序在init的时候,获取了程序的命令行,做了sha256计算,验证是不是正常启动的程序(sh,zsh,bash之类的,没有gdb).可能是为了反调试吧,所以在调试的时候,就用attach的方法了.

加密部分:

```

val.sival_int = 0x9E3779B9;
v0 = getppid();
sigqueue(v0, 34, val);
usleep(0x3E8u);
val.sival_int = 1113939753;
v1 = getppid();
sigqueue(v1, 35, val);
usleep(0x3E8u);
val.sival_int = -1635635460;
v2 = getppid();
sigqueue(v2, 35, val);
usleep(0x3E8u);
val.sival_int = -634942318;
v3 = getppid();
sigqueue(v3, 35, val);
usleep(0x3E8u);
val.sival_int = 1312119394;
v4 = getppid();
sigqueue(v4, 35, val);
usleep(0x3E8u);
for ( i = 0; i <= 6; ++i )
{
    read(0, (void *) (qword_603308 + 8 * i), 8uLL);
    vala.sival_ptr = (void *) qword_603308;
    v5 = getppid();
    sigqueue(v5, 36, vala);
    usleep(0x3E8u);
    for ( j = 0; j <= 31; ++j )
    {
        v6 = getppid();
        kill(v6, 37);
        usleep(0x3E8u);
        v7 = getppid();
        kill(v7, 38);
        usleep(0x3E8u);
        v8 = getppid();
        kill(v8, 39);
        usleep(0x3E8u);
    }
    v9 = getppid();
    kill(v9, 40);
}

```

可以看到,在输入前,程序先初始化了一些参数.

这是设定的接收函数.

```

v3 = __readfsqword(0x28u);
sigemptyset(&v1.sa_mask);
v1.sa_handler = (__sighandler_t)sub_4019EA;
v1.sa_flags = 4;
sigaction(34, &v1, &v2);
sigemptyset(&v1.sa_mask);
v1.sa_handler = (__sighandler_t)sub_401A09;
v1.sa_flags = 4;
sigaction(35, &v1, &v2);
sigemptyset(&v1.sa_mask);
v1.sa_handler = (__sighandler_t)sub_401A3A;
v1.sa_flags = 4;
sigaction(36, &v1, &v2);
v1.sa_handler = (__sighandler_t)sub_401AA4;
v1.sa_flags = 0;
sigaction(37, &v1, &v2);
v1.sa_handler = (__sighandler_t)sub_401AF6;
v1.sa_flags = 0;
sigaction(38, &v1, &v2);
v1.sa_handler = (__sighandler_t)sub_401B11;
v1.sa_flags = 0;
sigaction(39, &v1, &v2);
v1.sa_handler = (__sighandler_t)sub_401B66;
v1.sa_flags = 0;
sigaction(40, &v1, &v2);
v1.sa_handler = (__sighandler_t)sub_401B9D;
v1.sa_flags = 0;
sigaction(41, &v1, &v2);
return __readfsqword(0x28u) ^ v3;

```

分析

这部分来自做题的时候,零碎的分析.

0x603308 共享的内存地址 也是输入的区域

SIG 信号分析

准备工作:

34, 四次35

int[4] table (295f6542fc2e829e928c27da625a354e)

0x60324C : group_count

36 : sub_401A3A 将输入分2组,一组4字节,第一组603310 第二组603300 计数器+=2

进行32次 37,38,39

一次40

输入完成后,41

最终check : dword_603280

37:

First_Part += (table[dword_603244&0b11]+dword_603244) ^ (((LastPart>>5)^(16*LastPart))
+ LastPart)

38:sub_401AF6

dword_603244+=0x9E3779B9

39:

LastPart+= (table[(dword_603244 >> 11) & 3] + dword_603244) ^ (((First_part >> 5) ^ (16 *
First_part)) + First_part)

```
40:401b66
dword_603280[Group_count-2]=First_Part
dword_603280[Group_count-1]=LastPart
```

被37-39这段运算吓到了,觉得不好手动解密,从网上搜了半天,终于找到了这个算法:xtea.

尝试用python解密,但结果不正确.

又从网上搜索了c的代码,尝试一下解密第一组,结果正确.

改造了一下程序,解密了密文,得到了flag.

C代码

```
void decipher(unsigned int num_rounds, char * v, uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0 = *(uint32_t*)(v), v1 = *(uint32_t*)(v+4), delta = 0x9E3779B9, sum = delta * num_rounds;
    printf("%d %d\n", v0, v1);
    for (i = 0; i < num_rounds; i++) {
        v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
        sum -= delta;
        v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
    }
    *(uint32_t*)(v) = v0; *(uint32_t*)(v+4) = v1;
}

int main()
{
    char secret[99];
    FILE* fd;
    fopen_s(&fd, "d:\\t\\secret", "rb");
    fread_s(secret, 57, 56, 1, fd);

    uint32_t k[4] = { 1113939753, -1635635460, -634942318, 1312119394 };
    // v为要加密的数据是两个32位无符号整数
    // k为加密解密密钥,为4个32位无符号整数,即密钥长度为128位

    for (int i = 0; i < 7; i++) {
        decipher(32, (secret + i * 8), k);
    }
    secret[56] = 0;
    printf("解密后的数据: %s\n", secret);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
665438441 -1163299916
-1430228231 -1543864320
-638270765 -79729723
-741090978 -2036982409
-436157121 -1732499527
162186156 1792812068
101049092 1709673591
解密后的数据: hgame{No_ONe_c4N_t0_Be_@_$taR.Th3y_st11L_CAn_c4N_sH1n3.}
```

密得到flag

easyVM

结合网上资料,得知VM是指模拟了汇编运行, stack之类的。

程序设计了字节码,并定义了相应字节码的操作函数。

理想做法应该是分析出来操作函数,然后还原操作。

但xiaoyu说,这道题有一种可能会被出题人打死的解法,我开始另辟蹊径。

非预期解法？

- 1.题目是将输入的40个字节加密后与预期结果比较。
- 2.在主函数中，程序进行了一系列很有规律的赋值操作，大约是3*40次，每3个字节一组。这些组的2个字节都相同，只有另外一个字节不相同。
- 3.将不相同的字节导出成为char[40]，与预期结果xor，得到了flag? ! ? !

预期解法？

还没有用预期方法做。

待填。

END

四周的HGAME结束了，也知道自已有多菜了。

由于SARS-CoV-2的爆发，线下赛似乎有些遥远。

总之，希望能进入协会，和同学们一起做想做的事情。

--Mezone, 2020年2月15日 02点23分。

