常见加密算法

base64

A. 标准 base64
在 CTF 算法逆向中Base64 算是经常见到的一类编码。
Base64 编码之所以称为Base64，是因为其使用 64 个字符来对任意数据进行编码，同理有 Base32、Base16编码（Hex 编码）。标准 Base64 编码使用的 64 个字符为：

字符	值	字符	值	字符	值	字符	值
A	0	Q	16	g	32	w	48
B	1	R	17	h	33	x	49
C	2	S	18	i	34	y	50
D	3	T	19	j	35	z	51
E	4	U	20	k	36	0	52
F	5	V	21	l	37	1	53
G	6	W	22	m	38	2	54
H	7	X	23	n	39	3	55
I	8	Y	24	o	40	4	56
J	9	Z	25	p	41	5	57
K	10	a	26	q	42	6	58
L	11	b	27	r	43	7	59
M	12	c	28	s	44	8	60
N	13	d	29	t	45	9	61
O	14	e	30	u	46	+	62
P	15	f	31	v	47	/	63

Base64 编码本质上是一种将二进制数据转成文本数据的方案。对于非二进制数据，是先将其转换成二进制形式，然后每连续 6 比特（2 ^ 6 = 64）计算其十进制值，根据该值在上面的索引表中找到对应的 字符，昀终得到一个文本字符串。

假设我们要对`Hello!`进行 Base64 编码，其转换过程如下图所示：

原始字符	H								e								l								l								o								!							
ASCII码十进制值	72								101								108								108								111								33							
二进制值	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	1	0	1	1	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	1	0	0	0	0	1
Base64码十进制值	18								6								21								44								27								6							
Base64编码后字符	S								G								V								s								b								G							

可知 `Hello!` 的 Base64 编码结果为 `SGVsbG8h`，每 3 个原始字符经 Base64 编码成 4 个字符。那 么，当原始字符串长度不能被 3 整除时，怎么办呢?

以`Hello!!`为例，其转换过程为:

原始字符	H								e								l								l								o								!							
ASCII码十进制值	72								101								108								108								111								33							
二进制值	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0	1	0	1	1	0	1	1	0	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	1	0	0	0	0	1
Base64码十进制值	18								6								21								44								27								6							
Base64编码后字符	S								G								V								s								b								G							
原始字符	!																																																		
ASCII码十进制值	33																																																		
二进制值	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0																												
Base64码十进制值	8								16								0								0																										
Base64编码后字符	I								Q								A								A																										

`Hello!!` Base64 编码的结果为 `SGVsbG8hIQAA`。可见，不能被 3 整除时会采用补 0 的方式来完成编码。

需要注意的是：标准 Base64 编码通常用 = 字符来替换昀后的 A，即编码结果为 。因 为 = 字符并不在 Base64 编码索引表中，其意义在于结束符号，在 Base64 解码时遇到 = 时即可知道 一个 Base64 编码字符串结束。

```
#include <stdio.h>
#include <stdlib.h>

#define uint8_t unsigned char

char base_table[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

char* b64_encode(uint8_t *bindata, int length)
{
    int i, j;
    uint8_t current;
    char *base64;

    base64 = (char*)malloc((length + 2) / 3 * 4 + 1);

    /*每次处理三个字符*/
    for ( i = 0, j = 0 ; i < length ; i += 3 ) {
        /*前6个bit 首先右移两位*/
        current = (bindata[i] >> 2) ;
        current &= (uint8_t)0x3F;
        base64[j++] = base_table[(int)current];
        /*第一个字节的最后两个二进制位*/
        current = ( (uint8_t)(bindata[i] << 4 ) ) & ( (uint8_t)0x30 ) ;
        if ( i + 1 >= length ) {
            base64[j++] = base_table[(int)current];
            base64[j++] = '=';
            base64[j++] = '=';
```

```
            break;
        }
        /*9~12bit, 并连接7~8bit*/
        current |= ( (uint8_t)(bindata[i+1] >> 4) ) & ( (uint8_t) 0x0F );
        base64[j++] = base_table[(int)current];
        /*13~16bit*/
        current = ( (uint8_t)(bindata[i+1] << 2) ) & ( (uint8_t)0x3C ) ;
        /*就此结尾*/
        if ( i + 2 >= length ) {
            base64[j++] = base_table[(int)current];
            base64[j++] = '=';
            break;
        }
        /*17~18bit, 并连接13~16bit*/
        current |= ( (uint8_t)(bindata[i+2] >> 6) ) & ( (uint8_t) 0x03 );
        base64[j++] = base_table[(int)current];
        /*19~24bit*/
        current = ( (uint8_t)bindata[i+2] ) & ( (uint8_t)0x3F ) ;
        base64[j++] = base_table[(int)current];
    }
    base64[j] = '\0';
    return base64;
}


int main() {
    char test[] = "Hello World!";
    char *encoded = b64_encode(test, sizeof(test));

    printf("encode content: %s\n", encoded);

    free(encoded);
    return 0;
}
```

识别特征：

- 编码表长度为 64
- 编码后的字节数组长度为 4 的倍数，且为编码前的 4/3 倍（变长）
- 编码得到的字符串常以等号（=）结尾

B. 变种 base64
思路：
修改编码表（以上述代码为例）动态生成 / 修改编码表
将编码表修改为：

```
char base_table[] =
"abcdefghijklmnopqrstuvwxyzO123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ+/";
```

在该编码表下得到编码 `zCN7zTJP3hj71C3BxSj72SuSnhQ=`，明文该怎么求？

RC4

在密码学中，RC4（Rivest Cipher 4）是一种流加密算法，密钥长度可变，它加解密使用相同的密钥，因此也属于对称加密算法，RC4 是有线等效加密（WEP）中采用的加密算法，也曾经是 TLS 可采用的 算法之一。
补充说明：序列密码（流密码）：
流密码也属于对称密码，但与分组加密算法不同的是，流密码不对明文数据进行分组，而是用密钥生成与明文一样长短的密码流对明文进行加密，加解密使用相同的密钥。也就是说，RC4不是对明文进行分组处理，而是字节流的方式依次加密明文中的每一个字节，解密的时候也是依次对密文中的每一 个字节进行解密。

(1) s盒初始化

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + KEY[i mod key_len]) mod 256
swap(S[i], S[j])
endfor
```

（2）密钥流生成

```
i := 0
j := 0
while GeneratingOutput
i := (i + 1) mod 256
    j := (j + S[i]) mod 256
swap(S[i], S[j])
send(S[(S[i] + S[j]) mod 256]) to stream endwhile
```

(3)加密

```
for i from 0 to plain_len
    plain[i] ^= stream[i] endfor
```

c语言实现

```
#include <stdio.h>
#include <stdint.h>

void rc4_init(uint8_t *s, uint8_t *key, uint32_t len) {
    int i, j = 0;
    char k[256];
    uint8_t tmp = 0;

    for (i = 0; i < 256; i++) {
        s[i] = i;
        k[i] = key[i % len];
    }

    for (i = 0; i < 256; i++) {
```

```
        j = (j + s[i] + k[i]) % 256;
        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
    }
}

void rc4_crypt(uint8_t *s, uint8_t *buf, uint32_t len) {
    int i = 0, j = 0, t = 0;
    uint32_t k = 0;
    uint8_t tmp;

    for (k = 0; k < len; k++) {
        i = (i + 1) % 256;
        j = (j + s[i]) % 256;

        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;

        t = (s[i] + s[j]) % 256;
        buf[k] ^= s[t];
    }
}

int main() {
    uint8_t S[256];
    uint8_t key[] = "test_key";
    uint8_t data[] = "Hello World!";

    rc4_init(S, key, sizeof(key) - 1);
    rc4_crypt(S, data, sizeof(data) - 1);

    for (int i = 0; i < sizeof(data) - 1; i++)
        printf("%02X ", data[i]);

    printf("\n");
    return 0;
}
```

识别特征：

- 该算法在初始化 s 盒的时候有一个循环次数为 256 次的 for 循环
- 之后根据密钥打乱 s 盒
- 最后遍历输入明文的每个字节，从 s 盒中取一个字节与之异或，完成加密

用python解密

```
    from Crypto.Cipher import ARC4
    key = b"test_key"
    cipher = bytes.fromhex("A0 E2 BA F4 B5 02 ED 9B 41 2F E6 23")
    rc4 = ARC4.new(key)
    plain = rc4.decrypt(cipher)
    print(plain)
```

TEA系列

TEA 是 Tiny Encryption Algorithm 的缩写，以加密解密速度快，实现简单著称。
TEA 算法每一次可以操作 64bit(8byte)，采用 128bit(16byte) 作为key，算法采用迭代的形式，推荐的迭代轮数是 64 轮，最少 32 轮。
为解决 TEA 算法密钥表攻击的问题，TEA 算法先后经历了几次改进，tea -> xtea -> xxtea。
TEA 系列算法中均使用了一个 DELTA 常数，但 DELTA 的值对算法并无什么影响，只是为了避免不良 的取值，推荐 DELTA 的值取为黄金分割数与 232 的乘积，取整后的十六进制值为 `0x9e3779b9` ，保证每一轮加密都不相同。但有时该常数会以减法的形式出现，`-0x61c88647 = 0x9e3779b9`，因此出现0x61c88647是也要注意

TEA

```
#include<stdio.h>
#include <stdint.h>
typedef uint32_t u32;
#define delta -0x543210DD
#define rounds 32

void encrypt(u32 * v , u32 * k) {
    u32 v0 = v[0] , v1 = v[1];
    u32 k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
    u32 sum = 0;

    for (int i = 0; i < rounds; i ++) {
        sum += delta;
        v0 += ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5) + k1);
        v1 += ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5) + k3);
    }

    v[0] = v0;
    v[1] = v1;
}

void decrypt(u32 * v , u32 * k) {
    u32 v0 = v[0] , v1 = v[1];
    u32 k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
    u32 sum = delta * rounds;

    for (int i = 0; i < rounds; i ++ ){
        v1 -= ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5) + k3);
        v0 -= ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5) + k1);
        sum -= delta;

    }
```

```
    v[0] = v0;
    v[1] = v1;
}

u32 key[] = {0x12345678, 0x23456789, 0x34567890, 0x45678901};
u32 cipher[] = {0x2E63829D, 0xC14E40OF, 0x9B39BFB9, 0x5A1F8B14, 0x61886DDE,
0x6565C6CF, 0x9F064F64, 0x236A43F6 , 0x7D6B};

int main() {

    for (size_t i = 0; i < sizeof(cipher)/sizeof(u32) - 1 ; i+=2)
    {
        decrypt(&cipher[i], key);
    }
    printf("%s", &cipher);
}
```

识别特征：

- 标准 DELTA 常数（魔数）
- 密钥为 16 字节（4 个 DWORD）
- 加密轮数为 16/32/64 轮
- 加密结构中存在左 4 右 5 移位及异或运算
- 加密结构中存在轮加/减相同常数的语句

XTEA

XTEA 是 TEA 的升级版，变量类型处理和 TEA 一样，主要是加密逻辑有所变化，对每轮加密的 key 的 选择也有所变化。

```
#include<stdio.h>
#include <stdint.h>
typedef uint32_t u32;
#define delta -0x61C88647
#define rounds 32

void encrypt(u32 * v , u32 * k) {
    u32 v0 = v[0] , v1 = v[1];
    u32 k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
    u32 sum = 0;

    for (size_t i = 0; i < rounds; i++)
    {
        sum += delta;
        v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + k[sum & 3]);
        v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + k[(sum >> 11) & 3]);
    }

    v[0] = v0;
    v[1] = v1;
}

void decrypt(u32 * v , u32 * k) {
```

```
    u32 v0 = v[0] , v1 = v[1];
    u32 k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
    u32 sum = delta * rounds;

    for (size_t i = 0; i < rounds; i++)
    {
        v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + k[(sum >> 11) & 3]);
        sum -= delta;
        v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + k[sum & 3]);
    }

    v[0] = v0;
    v[1] = v1;
}

u32 key[] = {0x00010203, 0x04050607, 0x08090A0B, 0x0C0D0E0F};
u32 cipher[] = {0xC11EE75A, 0xA4AD0973, 0xF61C9018, 0x32E37BCD, 0x2DCC1F26,
0x344380CC};

int main() {

    for (size_t i = 0; i < sizeof(cipher)/sizeof(u32) ; i+=2)
    {
        decrypt(&cipher[i], key);
    }
    printf("%s", &cipher);

}
```

识别特征：

- 同 TEA
- 加密结构中存在右移 11 位并 & 3 的运算

XXTEA

XXTEA 是 XTEA 的升级版，其实现过程比前两种算法要略显复杂些，加密的明文数据可以不再是 64bit（两个 32 位无符号整数），并且其加密轮数是由 n，即待加密数据个数决定的。

```
#include <stdio.h>
#include <stdint.h>

#define DELTA 0x9e3779b9
#define MX (((z>>5^y<<2) + (y>>3^z<<4)) ^ ((sum^y) + (key[(p&3)^e] ^ z)))

void xxtea(uint32_t* v, int n, uint32_t* key)
{
    uint32_t y, z, sum;
    unsigned p, rounds, e;

    if (n > 1)            // encrypt
    {
        rounds = 6 + 52 / n;
        sum = 0;
```

```
        z = v[n - 1];
        do
        {
            sum += DELTA;
            e = (sum >> 2) & 3;
            for (p = 0; p < n - 1; p++)
            {
                y = v[p + 1];
                z = v[p] += MX;
            }
            y = v[0];
            z = v[n - 1] += MX;
        } while (--rounds);
    }
    else if (n < -1)      // decrypt
    {
        n = -n;
        rounds = 6 + 52 / n;
        sum = rounds * DELTA;
        y = v[0];
        do
        {
            e = (sum >> 2) & 3;
            for (p = n - 1; p > 0; p--)
            {
                z = v[p - 1];
                y = v[p] -= MX;
            }
            z = v[n - 1];
            y = v[0] -= MX;
            sum -= DELTA;
        } while (--rounds);
    }
}

int main()
{
    uint32_t v[] = {
                0xE74EB323, 0xB7A72836, 0x59CA6FE2, 0x967CC5C1, 0xE7802674,
0x3D2D54E6, 0x8A9D0356, 0x99DCC39C,
                0x7026D8ED, 0x6A33FDAD, 0xF496550A, 0x5C9C6F9E, 0x1BE5D04C,
0x6723AE17, 0x5270A5C2, 0xAC42130A,
                0x84BE67B2, 0x705CC779, 0x5C513D98, 0xFB36DA2D, 0x22179645,
0x5CE3529D, 0xD189E1FB, 0xE85BD489,
                0x73C8D11F, 0x54B5C196, 0xB67CB490, 0x2117E4CA, 0x9DE3F994,
0x2F5AA1AA, 0xA7E801FD, 0xC30D6EAB,
                0x1BADDC9C, 0x3453B04A, 0x92A406F9 };
    uint32_t  k[4] = {1 , 2 , 3 ,4};
    int n = sizeof(v) / sizeof(uint32_t);
    printf("Decrypted cipher: ");
    xxtea(v, -n, k);
    for (int i = 0; i < n; i++) {
        printf("%c", v[i]);
    }
    return 0;
}
```

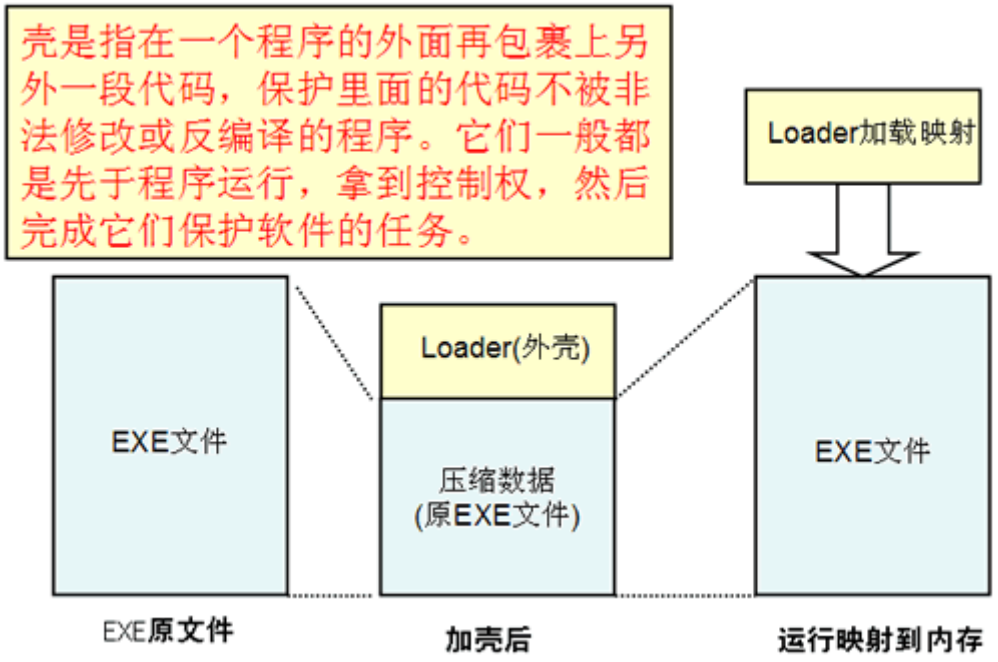
识别特征基本同 TEA，但是加密轮数通过计算求得（6 + 52/n）

逆向对抗

UPX 脱壳

认识壳

在计算机软件领域中有一类特殊的软件，它们专门负责对程序体积进行压缩，或者是保护一个程序不被非法修改、反编译、逆向分析，由于这类软件的作用类似自然界中的保护壳，我们便习惯性地称它 们为“壳”。
壳通过附加自身代码在保护对象（原始程序）上，在原始程序代码执行之前获得程序控制权，进而对原始程序进行解压或解密等还原操作。在这一系列操作结束之后，控制权将会被交还给原始程序，程序的原始代码将开始被执行。



壳的分类

一般根据壳的侧重方面将壳划分为两大类：

- 侧重于缩小程序体积，则被称为 压缩壳
- 侧重于程序代码及数据加密，则被称为 加密壳

壳特点

- 加壳程序初始化时保存各寄存器的值
- 外壳执行完毕，恢复各寄存器值
- 最后再跳到原程序执行

脱壳方法论

如今脱壳主要分为两种方式，第一种方式是：

工具脱壳，所谓脱壳机是针对特定的一种或一类壳开发出来的脱壳软件，它们是由他人在逆向完壳的 相关原理后编写的一类自动化工具，具有一定的局限性。

对于无法使用工具成功脱壳的情况，便需要用到第二种方式：

手动脱壳（手脱），也就是通过一步步分析程序加壳原理，手动还原原始程序的过程，通过手动脱 壳，我们可以加深对 PE 格式的理解，增长自己的脱壳水平。

在手脱的过程中，我们一般会涉及如下几个概念：

1. 查壳。遇到一个加壳程序，第一步应该去分析这是一个什么壳保护的程序，之后我们才能更加有针对 性地进行分析与跟踪，遇到难以处理的问题也可以更方便地利用搜索引擎检索同类壳的技术贴。常用 的查壳工具有 PEiD、Exeinfo PE、DIE 等
2. 寻找程序的原入口点（OEP）。通过单步跟踪、ESP 定律、内存断点等方法找到 OEP。
3. Dump 内存。所谓 Dump 内存，指的是将一个进程的内存镜像通过某种方式抓取下来，保存至本地磁盘中，之所以需要 Dump 内存，是因为在程序执行到 OEP 时，内存的状态往往就与未加壳前的程序相同（因为壳段代码已经帮我们完成了解密、解压等工作），此时我们将这个状态保存出去，再加上一些后期的修复，就能完成整个脱壳操作了。
4. 输入表（IAT）重建。在 Dump 操作结束之后，程序并不能直接运行，很大程度上是因为输入表并没 有被修复好。IAT 可以根据 PE 结构手动修复，也可以通过工具完成。
5. 关闭程序重定位

例题练习

花指令

花指令实质就是一串垃圾指令，它与程序本身的功能无关，并不影响程序本身的逻辑。在软件保护 中，花指令被作为一种手段来增加静态分析的难度，花指令也可以被用在病毒或木马上，通过加入花 指令改变程序的特征码，躲避杀软的扫描，从而达到免杀的目的。
花指令一般被分为两类：会被执行的和不会被执行的（垃圾指令）

会被执行

这类花指令本身是正常的汇编指令，它们运行完后不会改变原来程序的堆栈/寄存器，但能起到干扰静态分析的作用。一般分为两种：

- 形式一：改变堆栈操作
- 形式二：利用 call 指令或 jmp 指令增加执行流程复杂度

形式一

```
#include <stdio.h>
int main()
{
    _asm {
        push eax;
        add esp, 4;
}
printf("Hello World!\n"); }
```

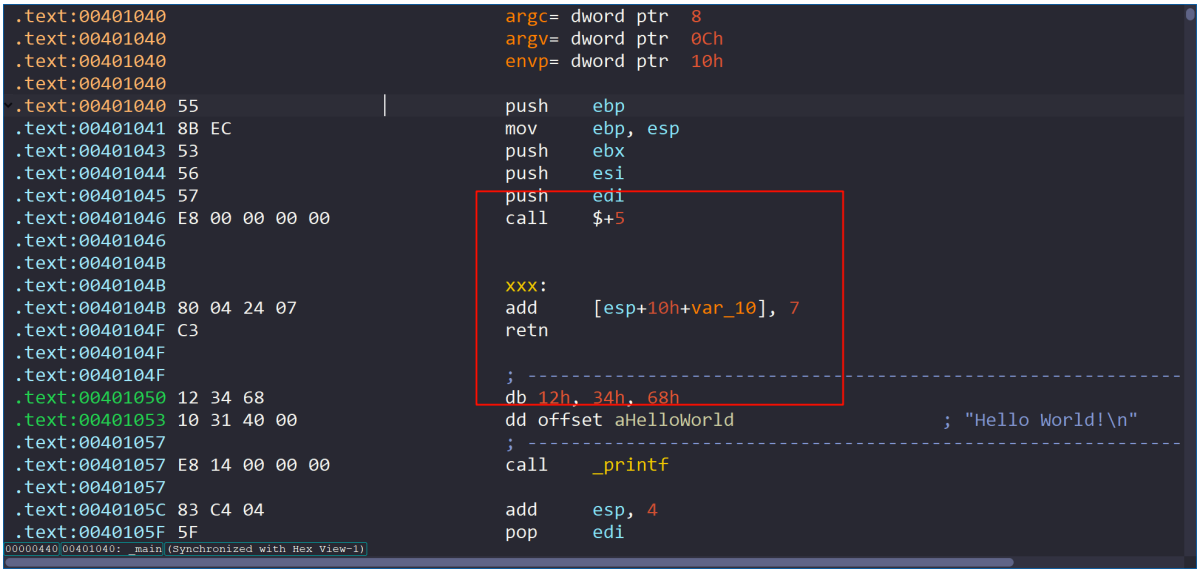
通过改变esp，让ida分析错误。但是现在ida能识别出来这种简单的花指令，对于现在版本这种部分失效。

形式二

执行 call 执行时会向堆栈中压入返回地址，我们可以修改这个返回地址，配合 ret 指令跳转到任意一 个想去的地方

```
#include <stdio.h>
int main()
{
    _asm {
        call xxx
    xxx:
        add [esp], 0x7
        retn
        _emit 0x12
        _emit 0x34
}
printf("Hello World!\n"); }
```

ida识别失败，误以为第一个ret是函数的结束。



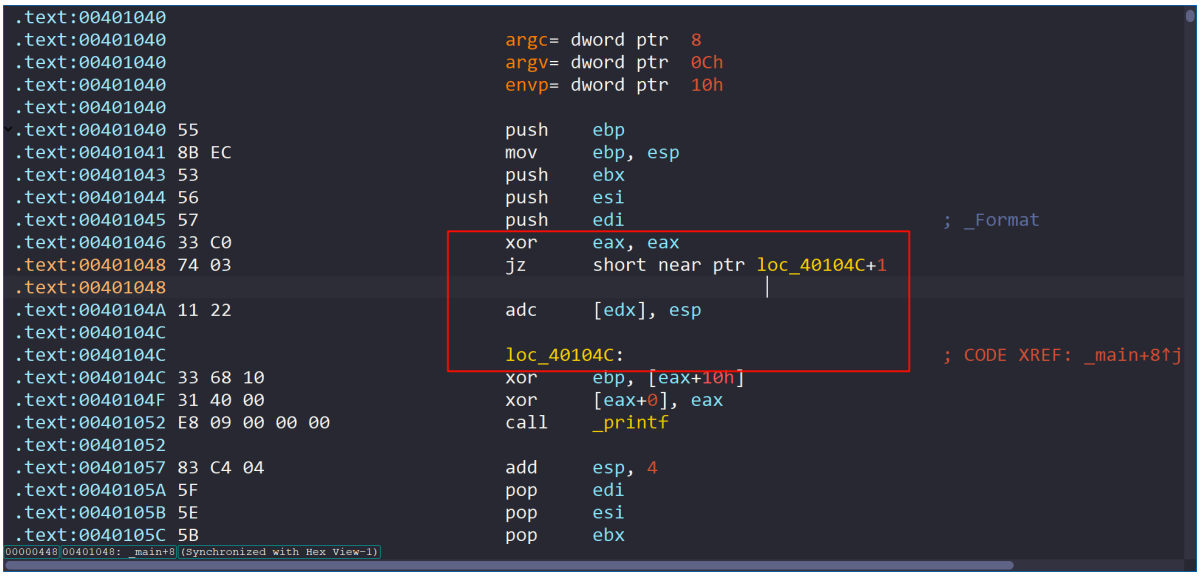
不会被执行

形式一

如果我们插入的花指令是一个操作码，那么后面程序原本的机器码就会被误认为是这个操作码的操作数，从而导致反汇编引擎的解析错误

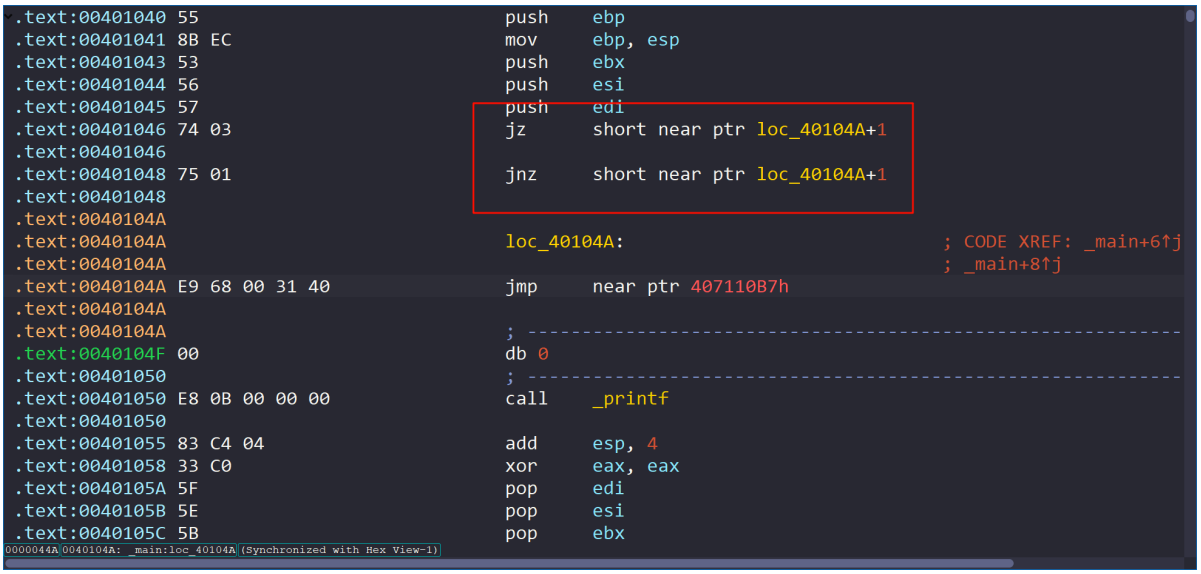
```
#include <stdio.h>
int main()
{
    _asm {
        xor eax, eax;
        jz xxx;
        _emit 0x11;
        _emit 0x22;
        _emit 0x33; // 0x33是 xor 指令的操作码，会导致后面正常的 push 指令被错误解析
    xxx:
    }
    printf("Hello World!\n");
}
```

ida识别为操作码，导致识别失败，但是由于经过xor 后，ZF 标志位被置为 1，那么 jz 这条跳转指令必定会被执行，后面插入的 0x11,0x22,0x33 就会被跳过，程序正常运行。



下面这种形式是最常见的

```
#include <stdio.h>
int main(int argc, char const* argv[]){
    _asm{
        jz label
        jnz label
        _emit 0xe9
        label:
    }
    printf("Hello world!\n");
    return 0;
}
```



形式二

插入的花指令也可以是改变堆栈平衡的汇编代码，跟形式一相同，在这些花指令上面写上跳转指令， 虽然花指令不会被执行，但是IDA进行解析时会认为该函数堆栈不平衡

```
#include <stdio.h>
int main(int argc, char const* argv[]){
    _asm{
        test eax,0          // 构造必然条件实现跳转，绕过破坏堆栈平衡的指令
        jz label
        add esp,0x1
        label:
    }
    printf("Hello world!\n");
    return 0;
}
```

总结

现在又三种情况ida无法识别

- 互补条件跳转（比较好处理）
- 永真条件跳转 （各种永真条件比较难匹配）
- call&ret跳转（比较难处理）

对抗思路

识别出来花指令，并NOP掉

例题练习

自修改代码

原理详解

自修改代码（Self-Modifying Code），指在一段代码执行前对它进行修改。把代码以加密的形式保存 在可执行文件中（或静态资源中），然后在程序执行的时候进行动态解密。这样我们在采用静态分析 时，看到的都是加密的内容，从而减缓甚至阻止静态分析。

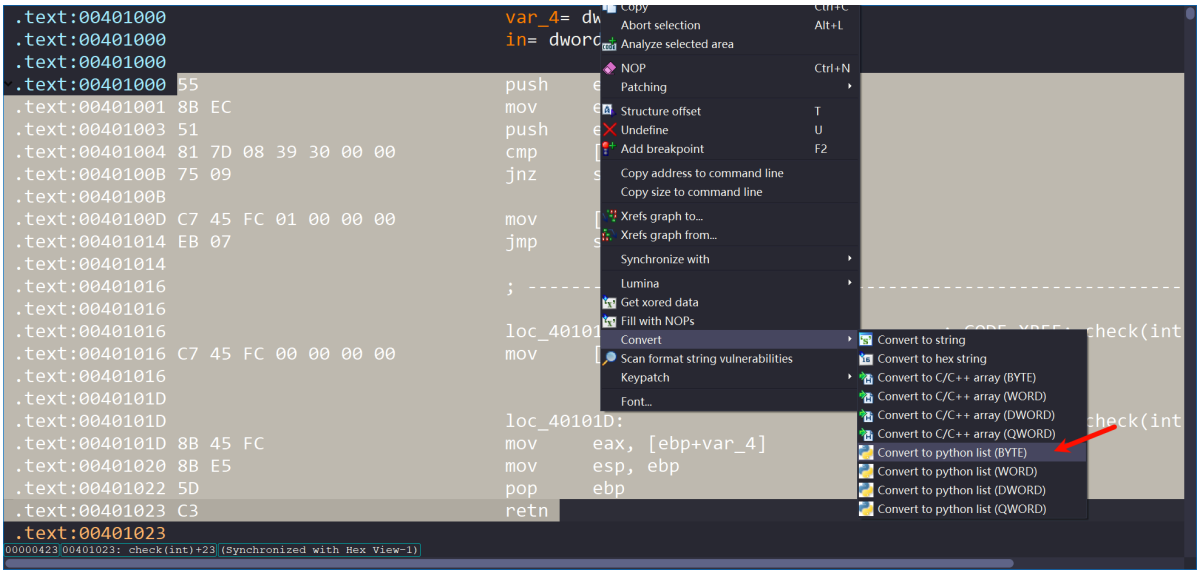
先来看一段展示 SMC 思路的伪代码：

```
if (运行条件满足) {
  DecryptProc(Address of Check) // 对 Check 代码解密
// ........
  Check();                      // 调用 Check
// ........
  EncryptProc(Address of Check) // 再对代码进行加密，防止程序被 dump
}
```

下面来打造一个使用 SMC 进行静态分析对抗的示例，首先正常写一个程序：

```
#include <stdio.h>
#include <windows.h>
int check(int in) {
    return in == 12345;
}
void decrypt() {
    DWORD old;
    VirtualProtect(check, 4096, PAGE_EXECUTE_READWRITE, &old);
    for (int i = 0; i < 36; i++)
    {
        *((char*)check + i) ^= 0x90;
    }
    VirtualProtect(check, 4096, old, NULL);
}
int main() {
    int input;
    scanf("%d", &input);
    decrypt();
    if (check(input))
        printf("good!");
}
```

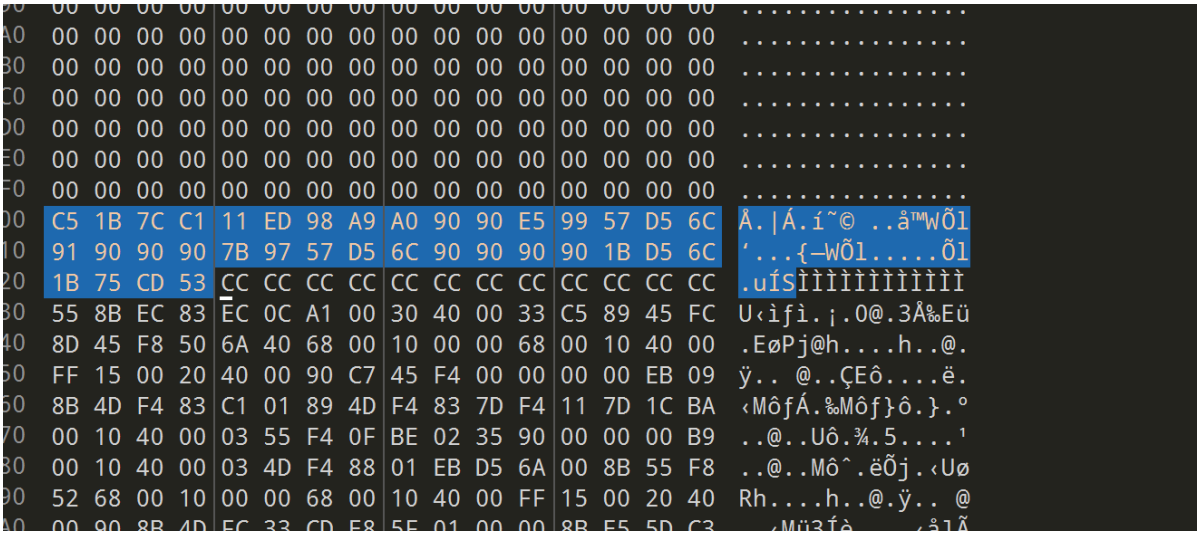
把check提取出来进行加密



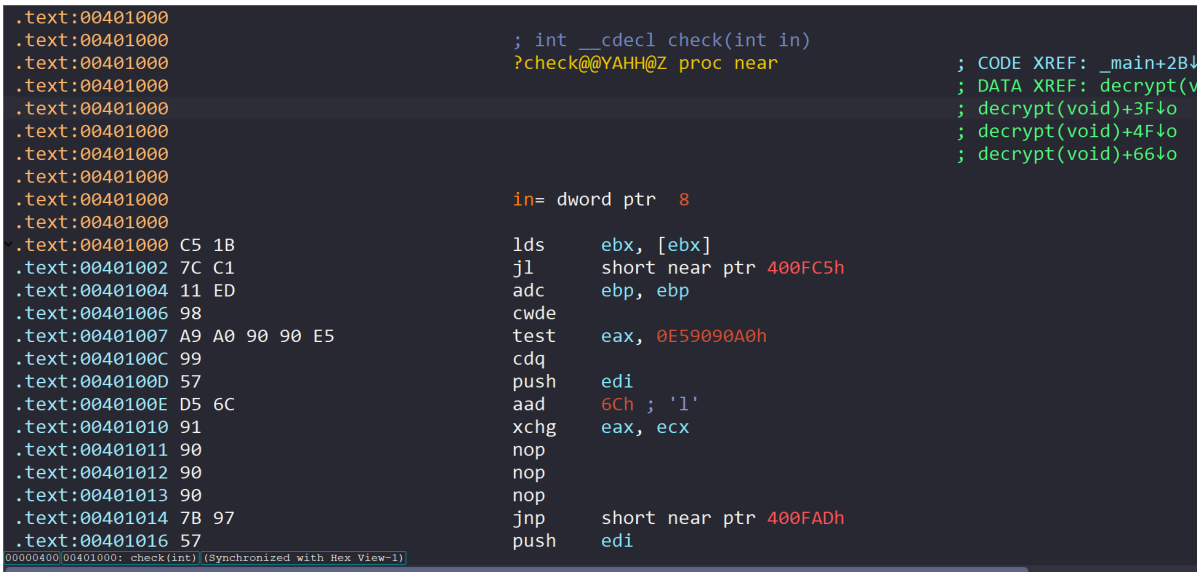
加密程序

```
check = [0x55, 0x8B, 0xEC, 0x51, 0x81, 0x7D, 0x08, 0x39, 0x30, 0x00, 0x00, 0x75,
0x09, 0xC7, 0x45, 0xFC, 0x01, 0x00, 0x00, 0x00, 0xEB, 0x07, 0xC7, 0x45, 0xFC,
0x00, 0x00, 0x00, 0x00, 0x8B, 0x45, 0xFC, 0x8B, 0xE5, 0x5D, 0xC3]
print(len(check))
for i in range(len(check)) :
    print(hex(check[i] ^ 0x90) , end= " ")
```

查看几个字节，需改源程序解密位数，编译用010打开并替换check函数



程序仍然可以正常运行，但是check函数已经进行加密



对抗思路

能动态调试最好直接动态调试，因为在程序运行的某一时刻，它一定是解密完成的，这时也就暴露 了，使用动态分析运行到这一时刻即可过掉保护。
其次是根据静态分析获得解密算法，写出解密脚本提前解密这段代码。 解密得到的机器码可以通过 IDAPython 的 patch_byte 接口很方便地写回

还原SMC代码的idapython模板如下：

```
import idc
import idaapi
import idautils

# ========== 用户配置 ==========
start_ea = 0x00401000  # 起始地址
end_ea   = 0x00401024  # 结束地址
key = 0x90             # 解密的异或密钥
# ============================

def decrypt_byte(byte, key):
    """解密函数"""
    return byte ^ key

def restore_smc(start, end, key):
    size = end - start
    encrypted_bytes = [idc.get_wide_byte(ea) for ea in range(start, end)]

    decrypted_bytes = [decrypt_byte(b, key) for b in encrypted_bytes]

    print(f"[+] 恢复字节范围: 0x{start:X} - 0x{end:X}")
    print(f"[+] 加密字节: {[hex(b) for b in encrypted_bytes]}")
    print(f"[+] 解密字节: {[hex(b) for b in decrypted_bytes]}")

    # 覆盖原字节
    for i, b in enumerate(decrypted_bytes):
        idc.patch_byte(start + i, b)

    # 删除旧反汇编
    idc.del_items(start, idc.DELIT_SIMPLE, size)
```

```
    # 重新创建指令
    ea = start
    while ea < end:
        insn_len = idc.create_insn(ea)
        if insn_len == 0:
            print(f"[!] 无法反汇编 0x{ea:X}")
            ea += 1
        else:
            ea += insn_len

    print("[+] SMC 区域还原完成 ✅")
    if not idaapi.add_func(start, end):
        print(f"[!] 添加函数失败: 0x{start:X}")
    else:
        print(f"[+] 函数已自动创建: 0x{start:X} - 0x{end:X}")

# 执行脚本
restore_smc(start_ea, end_ea, key)
```