对字符串加解密的处理（七）

一、引言

这一篇里，我们讨论字符串加密的源起与发展，它会让我们更好的理解这个话题。以及引导我们找到解决它的新思路。

二、字符串加密的前身

在给字符串赋值时，除了使用字符表示，很多语言都支持直接使用码点表示，比如 a 即 61 。如下 str1 和 str2 都输出 `Hello world` 。

C | 复制代码

```c
#include <stdio.h>

int main() {
    char str1[] = {0x48,0x65,0x6c,0x6c,0x6f,0x20,0x77,0x6f,0x72,0x6c,0x64,0};
    char str2[] = "Hello world\0";
    printf("%s\n", str1);
    printf("%s\n", str2);
    return 0;
}
```

str1 相较于 str2 ，可读性较差，这种转换我们一般叫它字符编码。

对于 JavaScript 等常以源代码形式发布的语言来说，这么做可以对分析者带来一定的困扰。但对于 C/C++ 等编译后运行的语言来说，这么做是不够的。分析者通过反编译器对程序进行查看，智能的反编译器可以轻松识别和展示这些被编码的字符串。

比如 IDA，它通过启发式的办法检测栈上字符串，然后用语义等价的库函数展示它，最常使用的是 `strcpy` 和 `memcpy` 。我们上面的代码，编译成 SO 文件再拖到 IDA 里，反编译效果如下。

```
__int64 test(void)
{
  char v1[16]; // [xsp+18h] [xbp-28h] BYREF
  char v2[16]; // [xsp+28h] [xbp-18h] BYREF
  __int64 v3; // [xsp+38h] [xbp-8h]

  v3 = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
  strcpy(v2, "Hello world");
  strcpy(v1, "Hello world");
  printf("%s\n", v2);
  printf("%s\n", v1);
  return 0LL;
}
```

为了阻挠 IDA 的这种分析过程，可以像下面这样做处理。

```
int unknow(int num){
    return ((num + 100) - 100);
}

void test(){
    char str2[0x10];
    str2[0] = 0x48;
    str2[1] = 0x65;
    str2[2] = unknow(0x6c);
    str2[3] = unknow(0x6c);
    str2[4] = 0x6f;
    str2[5] = 0;
    printf("%s", str2);
}
```

IDA 反编译效果如下，实现了字符串的顺利隐藏。

```
__int64 test(void)
{
  char v1[16]; // [xsp+18h] [xbp-18h] BYREF
  __int64 v2; // [xsp+28h] [xbp-8h]

  v2 = *(_QWORD *)(_ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2)) + 40);
  v1[0] = 72;
  v1[1] = 101;
  v1[2] = unknow(108);
  v1[3] = unknow(108);
  v1[4] = 111;
  v1[5] = 0;
  return printf("%s", v1);
}
```

应该说，字符编码所能提供的保护有限，很快就被更好用的字符串加密方案取代。

三、字符串加密的发展

在较早期，字符串加密这件事由开发者手动处理，比如像下面这样的代码

```
char* string = Base64Decode("aGVsbG8=");//hello
printf("string is: %s", string);
```

处理步骤如下

- 选择任意的加密或编码算法，从异或、Base64、Rc4 到 AES、SM4、RSA 等等都是可行的，只要加密或编码后，可以做对应的解密或解码即可，所以除了哈希算法基本都可以，这里我选择 Base64。
- 源码中添加 Base64 解密函数 Base64Decode
- 假设程序需要使用 `hello` 字符串
- 通过 cyberchef 或其他加密工具获取 `hello` 字符串在 Base64 编码后的结果 aGVsbG8=
- 使用 Base64Decode("aGVsbG8=") 表示 `hello` ，即在运行时计算出明文 `hello`

完整流程之后，源码就找不到 hello 字符串了，取而代之的是 aGVsbG8= ，这就是最朴素的字符串加密形式。读者可能会困惑，像下面这样不就节省了第四步，那么你明文隐藏了吗?

```
char* string = Base64Decode(Base64Eecode("hello"));
printf("string is: %s", string);
```

这个字符串保护方案虽然朴素好懂，但操作起来有些麻烦，每个字符串在使用时都要手动计算出对应的密文，并使用 Decrypt(xxx) 这样的方式替换使用，如果想为某个旧的项目添加字符串保护，就需要手动对项目中数以千百计的字符串逐一手工处理和替换，这从工程实践的角度上看，完全不能接受。

为了节省时间，简化步骤，一个朴素的思路诞生了 —— 将代码看作字符串，使用正则匹配和替换，实现之前手工所作的操作。为了减少误判，其中一些方案还需要将待加密的字符串记录到某个 XML 文件里，以方便后续做解析。

这个办法怎么看都有些粗糙和简陋，但它确实被应用过。读者可能会突然想起，我们前文采取过类似的字符串解密思路——正则匹配反编译后的伪代码，匹配密文和相关参数。换句话说，最早最朴素的字符串加密保护，和最早最朴素的字符串解密方案，它们在思路上完全一致。读者可能会进一步意识到，那么更先进的字符串保护方案，是否也可用于字符串解密处理？事实上确实如此，我们继续往下聊。

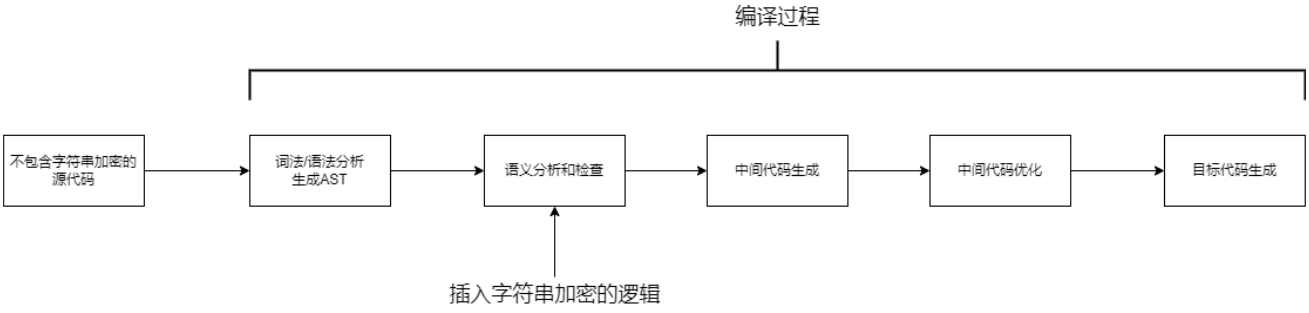
将源代码看作字符串实在太蠢了，为什么不把源代码转成抽象语法树（AST），然后在语法树层面做字符串的加密替换，最后将抽象语法树转回源代码呢？这个更灵活强大的改良方案很快就替代了原先的方案。

熟悉 JS 代码混淆和反混淆的读者对 AST 以及这套流程一定很熟悉，比如 Babel 库就是操纵 Javascript AST 的有力工具。对于 Native 语言，GCC、LLLVM 等编译器可以很好的生成语法树、处理并转回源码。

不论是将代码看成文本去做正则匹配，还是将代码转成 AST 操作完后回转源码，它们都属于代码编译前的方案，我们称之为第一类时机——编译前的保护方案。

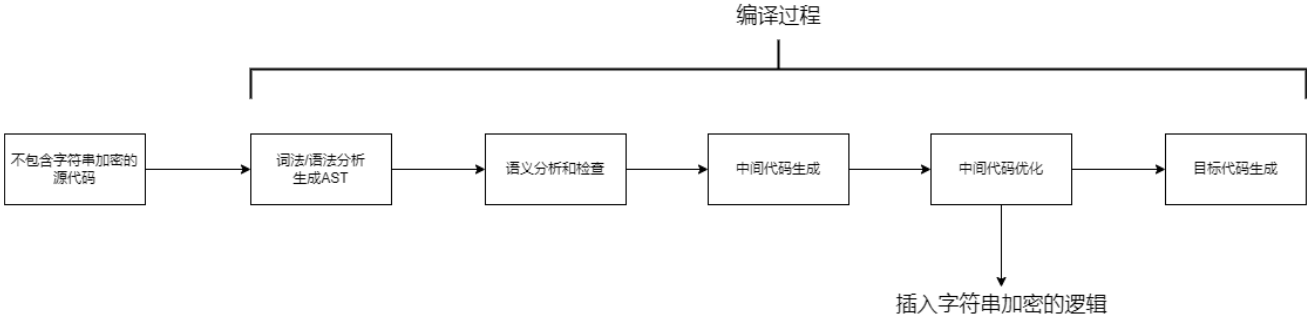
一些研究人员认为，编译过程本身就要做词法分析、语法分析、转换语法树这些步骤，因此完全没必要提前做字符串加密，而是应该由编译器去做包括字符串加密在内的种种工作。这就是第二类时机——编译期保护。

比如编译期通过操作语法树实现字符串加密



可以发现，相较于编译前操纵 AST 做处理的方案，它节省了重复转 AST 以及回转源码的步骤，但前者的好处是更灵活，你可以使用工具 A 完成字符串加密，使用工具 B 完成代码编译。

研究人员进一步发现，应该将处理时机放更后面的中间代码优化中，虽然在处理上更加抽象，但因为中间代码具有语言无关的特性，所以在此时机所实现的代码保护方案，通用于各种语言。



OLLVM 就是此类型的代码保护项目，它的处理时机就是在中间代码优化过程中对 IR 做处理，实现了控制流平坦化，指令替换等多种保护方案。

值得一提的是，OLLVM 并没有提供字符串保护，基于 OLLVM 改良的 Armariris 以及 Hikari 则提供了字符串加密保护，我们在前面的例子里和它们打过交道。

Armariris 会在 init_array 中添加一些函数，这些函数会对所有加密的字符串做解密，大家可能对它解密函数的函数名很熟悉，即 datadiv_decodexxx ， Armariris 的源码中对它命名的逻辑如下

Python | 复制代码

```
Constant* c = mod->getOrInsertFunction(".datadiv_decode" + random_str, FuncTy)
```

而 Hikari 采用了更复杂的运行时字符串解密，在每个函数的开头插入一个基本块，进行这个函数所涉及到的字符串的解密工作。假设原函数是 main。

Python | 复制代码

```
#include<stdio.h>
int main(){
    printf("Hello");
    return 0;
}
```

字符串加密后

Python | 复制代码

```
#include<stdio.h>
int main(){
    xor_decrypt(p_hello);
    printf(p_hello);
    return 0;
}
```

由于每个函数可能调用多次，而解密只需要调用一次，所以还需要一个开关，像下面这样。

Python | 复制代码

```
#include<stdio.h>
decryptStatu = 0;
int main(){
    if(decryptStatu == 0){
        xor_decrypt(p_hello);
    }
    decryptStatu = 1;
    printf(p_hello);
    return 0;
}
```

以上就是在编译期做字符串加密的简单概述。除此之外也可以在第三类时机——编译后的时机去做这件事。比较典型的就是解析二进制文件，找到数据段并整体加密，然后添加一个新段包含对应的解密函数，接着在 init_array 里调用解密函数，那么包括字符串在内的各种 data 段数据就被静态加密 + 运行时解密了。

上面我们描述了字符串加密所发生的三大类时机——编译前、编译中、编译后。

编译前：将源代码看作字符串正则匹配、用 AST 处理源代码

编译期：用 AST 或 IR 处理

编译后：解析二进制文件或整体加固

对于 Javascript / Python 等语言，因为不存在编译过程，常常以源码示人，所以混淆和反混淆主要基于编译前的 AST 方案。对于我们所关注的 Android Native 保护与对抗，三类时机的处理都很常见。

回到我们最关心的问题——字符串解密是不是也能用更花哨和强大的手段，而不仅仅是粗鄙的正则匹配？这是毫无疑问的，下一篇我们开始讨论这个问题。

四、字符串加密的具体方案

这一节我们回顾和总结前文所讨论的各种样本。如何评价和处理一个字符串加密方案？主要看以下几点

解密时机

- 在极其早的时机对全体加密字符串做解密
- 在函数起始处对当前函数使用到的全部字符串做解密
- 在字符串被使用前才做解密

解密算法

- 简单异或和编码
- 复杂算法

解密算法形式

- 内联汇编
- 单独函数

解密后明文存储方案

- 段
- 堆
- 栈

以上述四点衡量和评估我们所分析的各种样本

4.1 pdd_secure

解密时机是函数起始处

Shell | 复制代码

```
__int64 __fastcall sub_2D9B4(__int64 a1)
{
    unsigned int v1; // w8
    int v2; // w8
    int v4; // w8
    _BYTE v6[96]; // [xsp-60h] [xbp-D0h] BYREF
    __int64 v7; // [xsp+0h] [xbp-70h]
    unsigned __int64 v8; // [xsp+8h] [xbp-68h]
    int v9; // [xsp+10h] [xbp-60h]
    unsigned int v10; // [xsp+14h] [xbp-5Ch]
    __int64 v11; // [xsp+18h] [xbp-58h]

    v7 = a1;
    v8 = _ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2));
    v11 = *(_QWORD *)(v8 + 40);
    v1 = __ldar(dword_699B0);
    v9 = 527729684;
    v10 = v1;
    while ( 1 )
    {
        while ( 1 )
        {
            while ( v9 > -752143043 )
            {
                if ( v9 == -752143042 )
                {
                    pthread_mutex_lock(&stru_699C0);
                    if ( __ldar(dword_699B0) )
                        v4 = -1065271286;
                    else
                        v4 = -1890910423;
                    v9 = v4;
                }
                else
                {
                    if ( v10 )
```

解密算法是简单异或 `byte_6225C ^= 0x5Eu;`

解密算法形式是内联汇编而非单独函数，单独函数是更简单的方案，因为方便 Hook 和 Call，而且可以通过查看高频的函数引用轻松定位到它们。

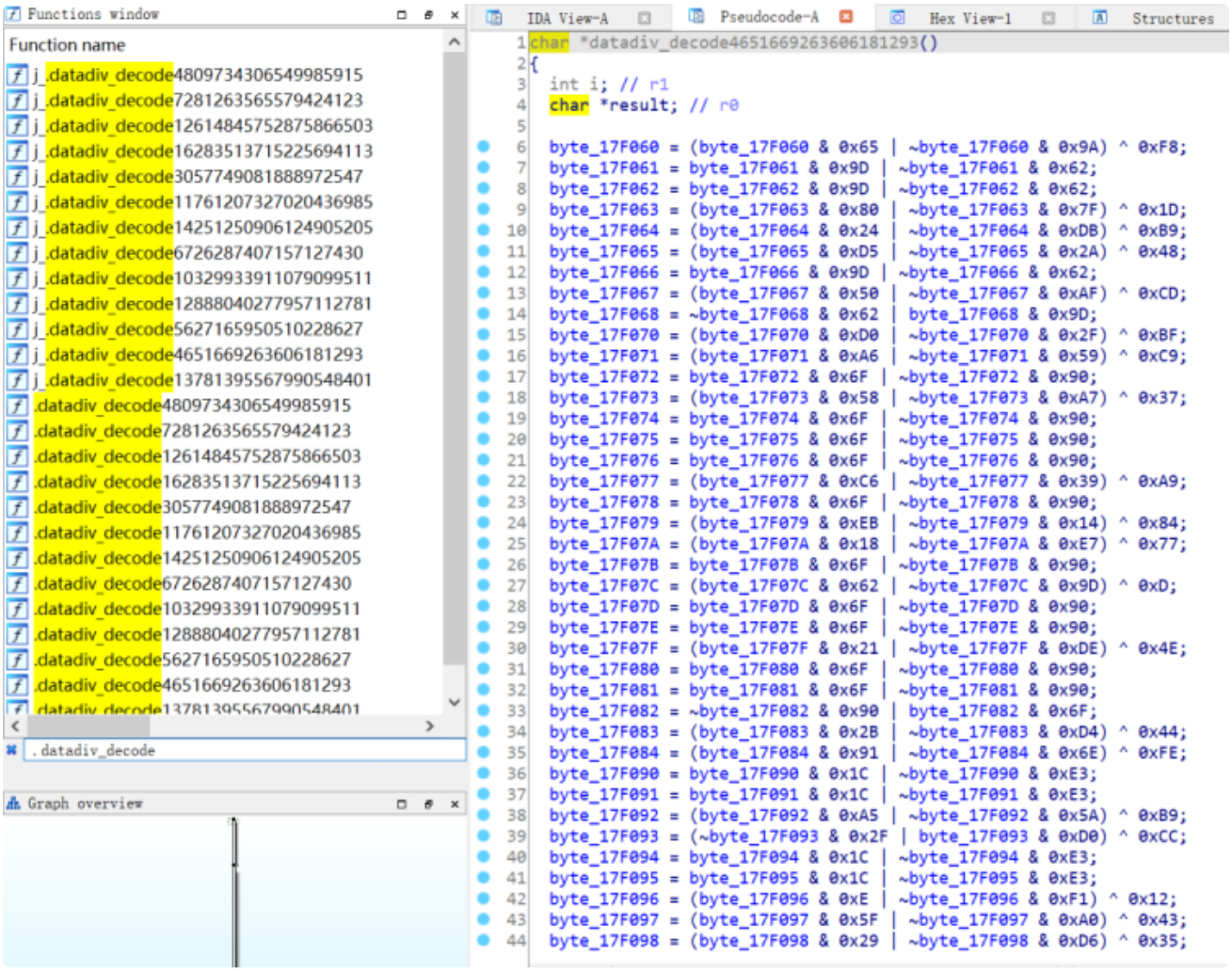
明文存储方案采用了覆盖密文的方式，因为密文原先在 data 段上，所以它属于段上存储。对于采用段上存储的方案，不论放 bss 段还是 data 段等等，又或者是否是覆盖解密，都可以通过 dump 方便处理。

4.2 net_crypto

解密时机是在极早处做全体解密

```
.init_array:00176BE8 ; Segment type: Pure data
.init_array:00176BE8                 AREA .init_array, DATA
.init_array:00176BE8                 ; ORG 0x176BE8
.init_array:00176BE8                 DCD .datadiv_decode4809734306549985915+1
.init_array:00176BEC                 DCD .datadiv_decode72812635655794241 23+1
.init_array:00176BF0                 DCD .datadiv_decode12614845752875866503+1
.init_array:00176BF4                 DCD .datadiv_decode16283513715225694113+1
.init_array:00176BF8                 DCD .datadiv_decode30577490818889725 47+1
.init_array:00176BFC                 DCD .datadiv_decode11761207327020436985+1
.init_array:00176C00                 DCD .datadiv_decode14251250906124905205+1
.init_array:00176C04                 DCD .datadiv_decode6726287407157127430+1
.init_array:00176C08                 DCD .datadiv_decode10329933911079099511+1
.init_array:00176C0C                 DCD .datadiv_decode12888040277957112781+1
.init_array:00176C10                 DCD .datadiv_decode5627165950510228627+1
.init_array:00176C14                 DCD .datadiv_decode4651669263606181293+1
.init_array:00176C18                 DCD .datadiv_decode13781395567990548401+1
.init_array:00176C1C                 DCD sub_49190+1
.init_array:00176C20                 DCD sub_491CC+1
.init_array:00176C24                 DCD sub_49314+1
.init_array:00176C28                 DCD loc_493B4+1
.init_array:00176C2C                 DCD sub_49448+1
.init_array:00176C30                 DCD sub_49478+1
.init_array:00176C34                 ALIGN 8
```

解密算法依然是简单异或，因为使用了指令替换，所以看起来稍显复杂，以下面图里的代码为例。



第一行里，`a ^ b` 替换为 `((~a & r) | (a & ~r)) ^ ((~b & r) | (b & ~r))`，因为 b 是常数，r 是已知的随机数，所以编译器将后半部分 `((~b & r) | (b & ~r))` 直接优化计算出了结果，呈现出 `((~a & r) | (a & ~r)) ^ immediate` 的效果。

第二行里，`a ^ b` 替换为更简单的 `(a & ~b)| (~a & b)`，`b` 即 0x62，等价于 `byte_17F061 ^ 0x62`。

解密算法形式上是内联汇编，不应该理解为单独函数，单独函数指的是传入密文输出明文的那种函数结构。

明文存储方案上也采用了覆盖解密，和前一个样本完全一致，可采用 dump 处理。

4.3 mtguard

解密时机在总体上属于具体使用前解密。

C 复制代码

```c
// byte_E2280、byte_E2281 是开关，防止多次重复解密
if ( !byte_E2280 )
{
    sub_5124(byte_CF579, byte_CFAC0, 14, 18);
    byte_CF58B = 0;
}
byte_E2280 = 1;
if ( !byte_E2281 )
{
    sub_5124(byte_CF58C, byte_CFAE0, 15, 21);
    byte_CF5A1 = 0;
}
byte_E2281 = 1;
v11 = (*(int (__fastcall **)(int, int, _BYTE *, _BYTE *))(*(_DWORD *)a1 + 4
```

解密算法算是简单异或

Shell 复制代码

```shell
_BYTE *__fastcall sub_5124(_BYTE *result, _BYTE *a2, int a3, int a4)
{
  if ( a4 >= 1 )
  {
    do
    {
      *result = *a2 ^ a3;
      --a4;
      ++result;
      ++a2;
      a3 += 3;
    }
    while ( a4 );
  }
  return result;
}
```

解密算法形式是单独函数，这让我们可以方便的 Hook 和 Call。

明文存储方案上不属于覆盖密文解密，但明文也放在数据段上，所以是段上解密，可以 dump。

4.4 metasec_ml

解密时机，是具体使用前才解密。

解密算法，仔细分析也是简单异或。

解密算法形式是单独函数，但不太一样的地方是有多个解密函数。

明文存储方案，我印象里是 `malloc` 开辟的明文内存，所以是在堆上。

4.5 SMSDK

解密时机更像是使用前解密。

解密算法，在字符串加密这一议题上，属于复杂算法。

解密算法形式是单独函数。

明文存储方案上似乎也是覆盖解密，我不太确定，感兴趣的可以试试 dump。

4.6 JDMobileSec

解密时机属于使用前解密。

C | 复制代码

```c
int sub_13404()
{
  _BYTE *v0; // r4
  char v2[92]; // [sp+0h] [bp-70h] BYREF

  v0 = sub_ABB0(flt_26E70, 20);
  _system_property_get(v0, v2);
  free(v0);
  return atoi(v2);
}
```

解密算法是简单异或。

解密算法形式是单独函数。

明文存储形式是堆存储，通过 `malloc` 开辟的空间。

```
_BYTE *__fastcall sub_ABB0(float *a1, int a2)
{
  _BYTE *result; // r0
  _BYTE *v5; // r1
  int v6; // r2
  float v7; // s0

  result = malloc(a2 + 1);
  if ( a2 >= 1 )
  {
    v5 = result;
    v6 = a2;
    do
    {
      v7 = *a1++;
      --v6;
      *v5++ = ((int)(float)(v7 + v7) ^ 0xDE) + 34;
    }
    while ( v6 );
  }
  result[a2] = 0;
  return result;
}
```

五、总结

本节讨论了字符串加密的源起，字符串混淆和反混淆的基本发展思路以及对前面各种例子的盘点这三部分。既是对前面内容的总结，也是进入新篇章的过渡，起到承上启下的作用。