moectf2024 Reverse 方向 进阶指北！

0xcafebabe *其实是因为这篇写的太难了所以成了进阶指北（小声）* *QQ: 782816967*

位运算基础

位运算是计算机中一种基于二进制位的运算方式，它直接操作二进制位，而不是数值本身。以下是位运算的基础知识和常见运算：

基础位运算：

OR（或运算）(|)：将两个二进制数的对应位进行逻辑或操作，只要两个位中至少有一个为1，结果就为1。

AND（与运算）(&)：将两个二进制数的对应位进行逻辑与操作，只有两个位都为1，结果才为1。

XOR（异或运算）(^)：将两个二进制数的对应位进行逻辑异或操作，如果两个位相同则结果为0，如果不同则结果为1。

NOR（或非运算）：与OR运算相反，结果为两个位都为0时为1，否则为0。

NAND（与非运算）：与AND运算相反，结果为两个位都为1时为0，否则为1。

NOT（取反运算）(~)：将二进制数的每一位取反，即0变为1，1变为0。

数据加密与位运算：

通过位运算可以实现简单的数据加密和解密。

异或运算是常用的加密手段之一。因为异或运算具有一些特性，如交换律和结合律，以及任何数和自身的异或结果为0等，这些特性使得异或运算在加密中应用广泛。

例如，通过将明文和密钥进行异或运算，可以得到加密后的密文；再次将密文和密钥进行异或运算，就可以还原出原始的明文。

优先级注意事项：

位运算和其他运算（如加减乘除）的优先级不同，需要注意遵循运算符优先级规则，或者使用括号来明确运算顺序，以确保表达式的正确性。

位运算符之间也有优先级顺序，一般来说，NOT运算的优先级最高，其次是AND、OR、XOR等。但为了避免混淆，建议在表达式中使用括号明确优先级。

1. 了解最基础的位运算: or, and, xor, nor, nand, not

2. 了解数据如何通过位运算加密?

如:异或(^)的性质：a^a=0, a^b=b^a, (a^b)^a=a^(b^a), 0^a=a, a^b^a=b

3. 时时刻刻注意位运算和其他运算、位运算自己之间的运算优先级！

算法解密：有限域下的乘法逆元

有限域是数学中一种特殊的代数结构，它包含有限个元素，并定义了加法和乘法运算。有限域中的乘法逆元是一个非常重要的概念。

首先，让我们来理解一下乘法逆元。在普通的数学中，如果我们有一个数 x，它的乘法逆元是另一个数 y，使得 x 乘以 y 等于 1。换句话说，如果我们有 x * y = 1，那么 y 就是 x 的乘法逆元。

在有限域中，情况稍微复杂一些，但基本思想是相同的。有限域中的每个非零元素都有一个乘法逆元。具体地说，对于有限域中的元素 a，它的乘法逆元记作 a^(-1)，满足以下条件：

a * a^(-1) = 1

换句话说，a 乘以它的乘法逆元等于 1。

以有限域 GF(p) 为例，其中 p 是素数。在这样的有限域中，对于任意非零元素 a，它的乘法逆元可以通过求解下面的方程来得到：

a * a^(-1) ≡ 1 (mod p)

这里的“mod p”表示对 p 取模。求解这个方程，就能找到 a 的乘法逆元 a^(-1)。

希望这样能让你理解有限域中乘法逆元的概念！

如果有下面这个代码，你如何求解？（C语言代码）

a1 是unsigned char*数组

a1[k]是有限域（0x00 ~ 0xff）

```
for (int j = 0; j < 12; ++j)
    a1[j] = a1[j] * 17 + 113;
```

我们都知道，(a1[j] - 113) / 17这样肯定是不行的，因为/17是整除，直接会损失掉数据，这时候我们就有乘法逆元，求得17在 mod 256下的乘法逆元是241，很容易写得解密算法：

```
for (int j = 0; j < 12; ++j)
    a1[j] = (a1[j] - 113) * 241;
```

反调试（强烈推荐x64dbg进行动态调试！）

调试(Debugging)是软件开发中的一项关键任务，它是指识别、定位和解决程序中的错误或异常的过程。调试通常在程序无法按预期工作或产生错误时进行，通过观察程序的执行过程、变量的状态以及可能的错误信息，来找出问题的原因并进行修复。

调试的主要工具包括使用断点、单步执行、输出变量值等。通过这些工具，程序员可以逐步地追踪程序的执行过程，找出错误所在，并逐步修复它们，直到程序能够按预期工作为止。

反调试(Anti-Debugging)则是一种技术，用于阻碍或干扰对程序进行调试的过程。这种技术通常由软件开发者或者恶意软件作者使用，他们希望阻止他人对程序进行逆向工程或者分析，从而保护程序的安全性或者隐藏其内部实现。

Linux 下的反调试讲解

在 Linux 下，反调试技术的实现与平台相关，并且可以使用各种不同的方法来检测调试器的存在。下面是一些常见的 Linux 反调试技术：

检查 /proc/self/status 文件： 在 Linux 中，每个进程都有一个对应的 /proc/[pid]/status 文件，其中包含了该进程的状态信息，包括是否被调试。反调试代码可以通过检查这个文件来确定是否被调试。

检查 ptrace 系统调用： 调试器通常会使用 ptrace 系统调用来与目标进程进行通信和控制。因此，反调试代码可以通过检查当前进程是否被其他进程使用 ptrace 进行跟踪来检测调试器的存在。

使用 fork 和 exec： 反调试代码可以通过 fork 一个新进程，然后使用 exec 载入目标程序，以检测调试器的存在。调试器通常会在 fork 和 exec 这样的系统调用中注入一些代码来跟踪目标进程。

信号处理： 调试器通常会使用信号来通知调试器当前进程的状态变化，比如断点触发或者单步执行。反调试代码可以通过捕获和处理这些信号来检测调试器的存在。

检查调试器相关的环境变量： 调试器通常会设置一些环境变量来控制调试过程。反调试代码可以通过检查这些环境变量来确定是否被调试。

反调试指令和技术： 反调试代码可以包含一些特殊的指令或者技术，用于检测调试器的存在。比如，一些反调试代码会在目标进程中注入一些不合法的指令，以触发调试器的异常处理。

Windows下的反调试讲解：

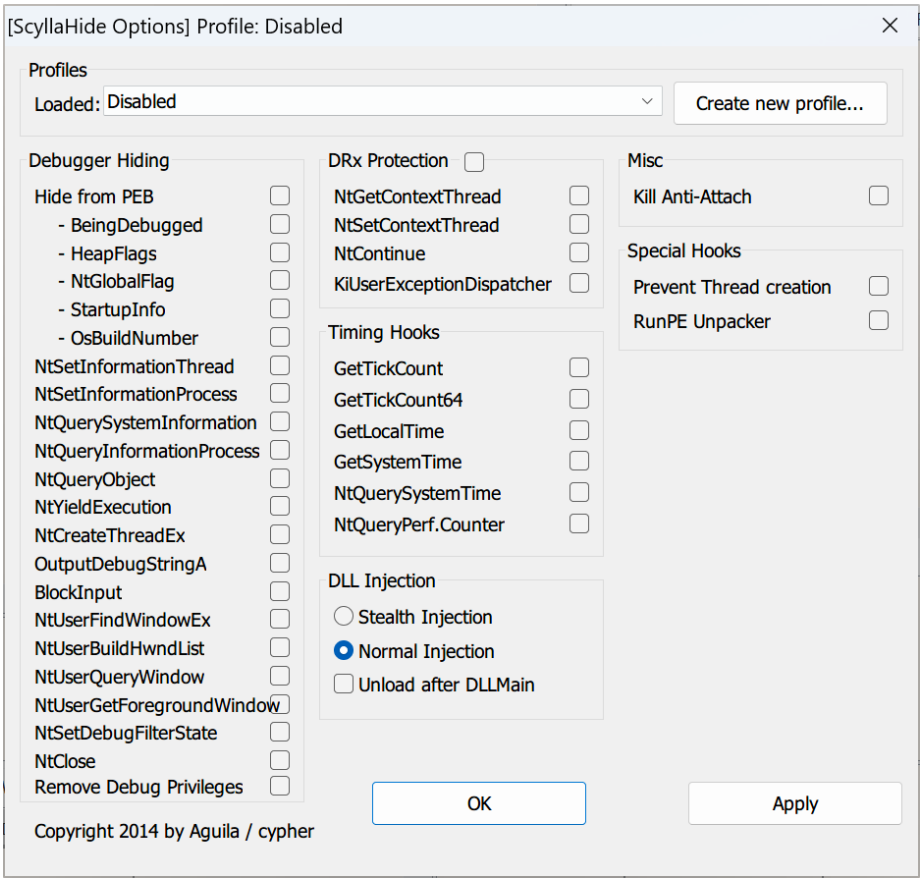
Windows操作系统相比于linux来说，复杂许多，因为要兼容先前的东西，反调试和调试对抗也非常激烈.jpg

反调试一般分为R0和R3下的，R0指的是内核(Kernel)层面，在这个层面下通过Hook SSDT表，注册Obj回调等高权限操作进行反调试与调试之间的打架，对于本次moectf2024，照顾到萌新们还要选择更多的方向，先不出这类的反调试。

R3反调试也就是应用层的，你打开的应用大多数都是跑在这个模式下的，这个模式的权限比较低，无法直接与硬件交流，要与硬件交流，需要调用windows system dll之中的函数，再通过这些函数syscall，进入windows内核，完成R3->R0的切换，当然，我们在R3下syscall函数过了就过了，你无法F7（单步进入）到里面。

R3下的反调试种类非常繁多，每一个都很简单，但是如果把它们组合起来，那么逆向难度将是巨大的，如果再加上汇编层面的代码混淆，代码虚拟化，那么无疑让人想把它丢入回收站。

R3下



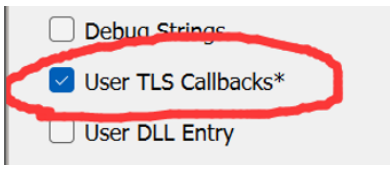
上图是x64dbg的一个反反调试插件（ScyllaHide），非常常用，建议读者家~~中常备，~~以免一些意外发生：有一些反调试，检测到你调试不给你爆，而是偷偷修改掉一些正确的数据或者是一些正确的代码逻辑，让你永远无法解密出flag 。

TLS下的反调试：

TLS 是 Thread Local Storage 的缩写，是一种线程局部存储的机制，用于在多线程程序中实现线程间的数据隔离。简单来说，TLS 允许每个线程都有自己独立的变量实例，这些变量只在当前线程内部可见，而不会被其他线程共享。

在反调试中，TLS 被用来实现一种技术，即在程序启动时，在 main 函数执行之前执行一些代码，通常是检测调试器是否存在或者进行其他反调试操作。这是因为 TLS 段的代码会在程序的任何其他代码执行之前优先执行。这样一来，即使在 main 函数设置断点，反调试代码也能够在主函数执行之前检测到调试器的存在。

TLS代码段会在程序main函数前优先执行，如果把反调试代码放在这个区域，那么你在main函数的断点甚至也能被检测出来！以防万一，建议读者在x64dbg的Options中勾选：



X64下的syscall 反调试：scyllahide原理是通过hook掉ntdll, kernel32dll, kernelbasedll中的相关函数，并在调用的时候返回正确值来对程序进行伪装，从而让程序无法判断正在被调试，从而绕过，而一些新型反调试直接在汇编中构造syscall，直接调用底层函数来检查反调试，这种情况的话，

ScyllaHide是无法进行反反调试的，可以试试SharpOD插件，如果SharpOD插件也不行，我们可以直接在内核进行对抗，使用TitanHide以及其插件，以上的这些插件，在github中均可搜到，读者可以慢慢学习！

注：本次比赛可能只会涉及到R3下的简单反调试，对于更深层次的逆天反调试，欢迎来戳我共同学习。

其他需要了解的：

1. X86/64汇编语言、ARM汇编（只需要认识，并且知道每条命令会让寄存器、内存进行如何变化，并且可以通过自己拿手的编程语言来“模拟”这些变化即可）。
2. X64dbg/IDA等工具的学习（只有工具用的顺，做题才做的顺）。
3. PE结构/ELF结构的学习（有助于你更深层次理解操作系统与二进制的关系）。
4. 掌握一门自己喜欢的编程语言，用于你来解密数据、写解密算法（优先推荐Python，其次是C/C++）。在一些位运算密集的算法中，我还是建议使用C/C++来写解密脚本，比较方便。
5. 必须掌握Python的一个库，叫做z3-solver，对于解决算法类难题有很大帮助（比如说解100个线性方程组 or 一个你看不明白<找不到漏洞的>如何逆向的哈希函数）
6. 逆向中不要分神，记得你抄出来一点算法，你就距离flag更进了一步。
7. 不要作弊，作弊可耻，moectf本意是给萌新一个接触、学习、掌握ctf的机会，而不是刷分上榜，作弊让👴抓到了可有你好🍎汁吃滴奥。

题目在下一页

⚠️别直接翻到这里来解flag！ ⚠️

⚠️moectf不会对先做出来的人有特殊奖励，你反而会失去一些学习的机会⚠️

写在最后：尝试自己解决这个问题，获得第一个flag吧！

```
void flag_encryption(unsigned char* input)
{
	size_t len = strlen((const char*)input);
	if (len != 44)
	{
		std::cout << "length error!";
		return;
	}
	unsigned int* p = (unsigned int*)input;
	for (int i = 0; i < 11; i++)
	{
		*(p + i) = (*(p + i) * 0xccffbbbb + 0xdeadc0de) ^ 0xdeadbeef + 0xd3906;
		std::cout << ", 0x" << std::hex << *(p + i) << ' ';
	}
	std::cout << std::endl;
	// 0xb5073388 , 0xf58ea46f , 0x8cd2d760 , 0x7fc56cda , 0x52bc07da , 0x29054b48 , 
0x42d74750 , 0x11297e95 , 0x5cf2821b , 0x747970da , 0x64793c81
}
int main()
{
	unsigned char a[] = "moectf{f4k3__flag__here_true_flag_in_s3cr3t}";
	flag_encryption(a);
	return 0;
}
```