# uctf信息安全_UCTF2016 twi Writeup

原创

weixin_39613561 于 2020-12-21 23:19:35 发布 127 收藏
文章标签： uctf信息安全


前言

该题目分类为REVERSE。最终目标是提交一个flag。

从题目来看，本题目应该是UCTF 2016中的题目，可能是由于笔者的搜索能力还有待提高，没有能够找到相关的Writeup。所以，本次只能自己摸着石头过河了。

解题准备

验证下载正确性

基础信息分析

文件类型识别

可以看到，文件是Atmel AVR 8-bit 的ELF文件。

查看是否有隐藏信息

从binwalk的结果来看，应该没有什么隐藏的数据。

字符串分析

各位小伙伴看到这个是不是就已经跃跃欲试了啊。反正我当时是觉得前途光明，形势非常好。不是小好，而是大好。。。

环境准备

模拟器

笔者手头并没有Atmel AVR架构的板子，所以优先考虑使用模拟器。

经过搜索，发现了三款模拟器：

simavr

simulavr

qemu-system-avr

考虑到simavr在网络上得到了比较多的推荐，优先考虑使用之。

笔者的系统是Ubuntu 20.04 LTS on Windows 10 WSL。这三款模拟器都可以通过apt 直接安装。

sudo apt install simavr

安装成功后，可以执行一下，看看说明信息。在说明信息中，发现了–list-cores参数。我们一并来看一下simavr都支持哪些core。

调试器

有了模拟器，我们可以执行镜像了。接下来，需要调试工具。

sudo apt install gdb-avr

gdb这种神器想必大家已经用得出神入化，不用笔者这种小白啰嗦啦。

反编译器

考虑到笔者对版权的洁癖以及贫穷，IDA虽然强大，但也只能忍痛放弃。

其实Ghidra已然足够强大了，我们本次选择ghidra_9.1.2_PUBLIC_20200212作为逆向工作环境。

在Ghidra中Import 目标文件，AVR架构可以正常识别。之后进行自动分析。

可见镜像从地址0x000000开始。由于镜像没有任何symbol或者debug信息，此处可能需要手动按D来强制进行反汇编。细心的小伙伴可能已经发现，明明每个指令是两个字节，但是前面的地址并没有按照两个字节增加。。。

Atmel AVR架构

Atmel AVR 8-bit 是一个精简指令集架构。具体信息大家可以去网上找。信息还是不少的，必定Arduino就是采用这个架构的。

AVR 采用哈弗结构，指令、数据分开寻址。从上面的图，大家可以看到code地址。其实还有一个mem地址。来看下图：

好的，我们现在来讨论一下上面说的，code地址的增加步进不是2的问题。

可以看到，mem寻址是按照1个字节的步进增加的。这里涉及到AVR的寻址模式。mem是按照正常的方式寻址，而code是按照2个字节为一组寻址。也就是内存中的地址是PC * 2。

关于AVR 8-bit的寄存器，总体来说和一般RISC架构的大同小异。不过这里要注意寄存器的别名。比如 W、X、Y以及Z，这个几个是用来间接寻址的寄存器。由于寻址需要，所以他们是8个通用寄存器，分成4对，作为4个16 bit寄存器。

需要注意的是，在gdb中执行“i r”的话，并不会显示这几个寄存器。为了能更方便的获取这几个寄存器的信息，笔者还专门写了一段gdb脚本。。。

在实际操作中，需要准备一份AVR 8-bit的指令说明书。笔者搞到这个里面里面带广告，就不放出来了，大家自己找找吧。

尝试运行

运行镜像，最大的挑战在于如何选择机型。笔者认为，越简单的东西，越不容易出问题。出了问题，也可以更方便的排查，所以先选择了atmega8 作为目标机型。当然，最终机型的确定是在整个解题过程中，一点点确定的。题目中的twi其实指的就是单片机中使用广泛的TWI协议，所以比如0x53、0x21和0x23肯定是TWI相关寄存器。另外，根据程序逻辑，判断0x60 肯定不能是寄存器，而是内存区域。

解题成功后，再次进行实验，目前发现atmega8 和 atmega32 都可以作为目标机型。

执行命令：

simavr -t -m atmega8 –gdb twi.70087b1e507aee08fc5c376a7b5ccc80

执行gdb，并attach：

avr-gdb

target remote :1234

这里笔者遇到了问题。尝试设定断点，可是得到：

首先，断点失败。这个在我偶然ptype $pc的时候找到了答案。由于AVR是指令数据分开寻址，所以要设定断点，我们需要提供code地址而不是data地址。怎么提供呢？大家自己ptype $pc下试试吧。当然，也可以参照笔者编写的gdb脚本。

其次，为啥明明设定的是0x26，地址却变成了0x800026。因为我们提供的0x26被认为是数据，而被模拟器自动转换到了对应的内存区域。具体情况大家可以参照simavr的源代码。

那么执行、断点的问题都解决了，我们可以开始进行逻辑分析了。

逻辑分析

入口点

通过Vector表，我们可以找到入口点：

这个函数是我自己加的，默认分析完这里是个label。这段代码的作用主要是进行栈的初始化，并且将程序段的一些数据copy到data区域以及进行一些内存区域的初始化。

从本块代码，我们可以看到，SP被初始化为0x45f。

接下来，将我们关注的字符串”flag{“ copy到了0x60。当然，还有”success”和”fail” 等等。然后跳转到了函数：FUN_code_0002d3。

FUN_code_0002d3

这个函数里面逻辑还是挺多的。

嗯，顿时头大。到这里，笔者决定先跑起来看看。

寻找线索

运行起来，没有Log输出。(后来发现，偶尔会有”fail” 的log出来，应该是模拟器的问题。)

经过了几次c – CTRL_C 循环后，发现程序都停在这里：

接下来看看这里在干什么。注意，这里地址是0x240，需要换算成code地址：0x120。

经过一番痛苦的分析，笔者发现：这段逻辑是个延迟。没有发现任何side effect，纯纯的延时。嗯，那还能咋办，为了能够进一步分析，当然要干掉啊。

这里，干掉这个延时有两个办法：

直接修改镜像，把无关紧要的代码修改为 xx c0或者 xx cf 跳转指令。

在gdb中下套。

当然，考虑到解题也是以学习为目的，笔者两个办法都尝试了。这里贴一下gdb下套的方案：

细心的盆友们可能又要问了，这里pc的设定为啥是按照内存地址值呢？看下一gdb中pc的显示：

注意，虽然显示的时候是按照code地址显示，但是大家在使用$pc设定值和运算的时候，一定要用内存地址值哦。

FUN_code_00012d

我们继续寻找线索。在把那个大延时干掉之后，我们又来到了一个新的区域。仍旧是c CTRL_C 循环，位置比较固定：

根据地址，寻找逻辑。

看到笔者对函数的命名，大家应该明白这个是干啥的了。这里，看起来好像很轻松就识别到了这个函数的作用，可是是实际上，笔者因此损失了多少本来就不多的头发啊！而且，靠的是灵光一闪，突然有思路。在无数次进入这个函数，笔者满地打滚、狂薅头发的过程中，突然想到：看看这东西到底在输出啥吧。。。

于是，这个函数的名字就有了。

接下来，肯定是要找引用点啦。

success 与 fail

在Ghidra强大引用分析的帮助下，不到1秒就成功定位相关逻辑。这样一来，程序的大体逻辑基本清晰了。另外，这两个点都在FUN_code_0002d3函数中。这样，就可以通过这两个指令块，来向上逆推正确的逻辑路径了。

确定路径是逆向中比较基本的工作，这里不再赘述。

第一个分支点

其中，0x60中保存的字符串是：”flag{\0”。

进入函数，分析其逻辑。

由于生平第一次接触AVR架构，再次经过了满地打滚以及狂薅头发的分析之后，发现这个函数是在检查W所指向地址中的内容前的5个字节是不是等于”flag{”。

在这个时候，笔者还没想到TWI的真实意义。(原谅我对硬件的无知与迟钝)，所以还以为程序运行到一定程度，会解密出flag放在内存中。于是，这里的解决方案是：将0x60的5个字节copy过来。。。虽然，结合谜底来看简直是蠢爆了，但是在最终解决问题阶段还是用上了，也算没白费劲儿。

(中式英语是伟大的！是她维护了文件编码的纯洁性。

— SimonTheCoder)

第二个分支点

接下来，来到LAB_code_00030a。

这段逻辑就比较容易理解了。在判断刚才那段内存中的0x26位置是不是”}”(0x7d)。

解决方案和上一个分支点的处理方法一致，你要什么，我就给什么。

数据处理函数

这里，涉及到两个函数(如图)。又是一番痛苦的分析之后，发现这两个函数一个是将flag{******}中的******(32个)向前移动5个字节，覆盖掉”flag{”。

另一个，将这32个字节作为hexstring转换为16个字节数据。

两个函数功能虽然简单，但是由于笔者对AVR不够熟悉，着实进行一番痛苦的分析。

最终分支

这个分支点是最后一个影响程序结果的分支点，也是最难搞的一个。

由于比较长，这里只截取了一小部分指令。这一大段逻辑，是用来对之前处理好的16字节数据进行校验的。注意是校验，而不是解密。

可以看到，最终是要判断R22和R10是否相当。根据逻辑，R10以及其它参与运算的寄存器的数据，都是从数据区域读出的固定值。而R22的基础值也是从固定数据中读出，与待验证的16个字节，经过运算后形成。

其中，负责提供算法复杂性的是FUN_code_00036c：

可以看到，大量寄存器参与了运算。并且还会进行mul运算。这么一段逻辑，估计要完全分析清楚估计要很长时间以及非常强大的精神力量。

另外，整段逻辑没有任何st指令。换言之，没有对内存进行写操作。这也就意味着，并不存在将数据解密后放在内存中的逻辑。笔者之前的猜测显然是错误的。

接受了这个事实之后，我们需要解决两个问题：

用来验证的flag是如何加载到内存中的。

flag的内容为何。

再一次的 FUN_code_0002d3

为了解决之前的两个问题，重新回到 FUN_code_0002d3进行分析。

用来验证的flag是如何加载到内存中的。

考虑到镜像文件中并没有相关的数据，那么只有可能是来自于外部输入。查看了一下FUN_code_0002d3中的函数调用：

几个函数中，只有000062我们目前还没有接触过。查看一下代码：

其实，只要分析清楚这块逻辑，就没有必要再进行000062的分析了。首先，可以看到这是个循环。该循环每次从FUN_code_000062中获得一个字节，放进Z中。而Z此时，正指向待check区域的地址(也就是flag放的地址)。(这个可以从gdb和使用了R3R2赋值两个角度来验证。)

len(“flag{}”)+len(“32字节神秘字符串”) = 0x6 + 0x20 = 0x26，count也对上了。

当然，实际上还是对FUN_code_000062进行了分析的。

节选一段。从代码来看，该函数再操作寄存器。根据寄存器，搜索了一下，发现是TWI相关寄存器。至此，题目中的TWI的来由我们清楚了。

从而，我们可以确定，本题目在板子上真机运行的时候，可以通过TWI将flag输入到程序中。

flag的内容为何。

确定了数据的来源，那么接下来解决数据的内容。

通过之前对数据校验区域的分析，我们可以得到一下结论：

算法非常复杂，短时间内难以完成你算法。

flag的核心是32字节的hex string，经过处理函数处理后，得到了16字节数据。

flag通过TWI 输入系统。

通过以上几个结论，再考虑到之前屏蔽掉的那个延时逻辑，那么解决方案就呼之欲出了，

出题人是在疯狂暗示我们：爆破。

PWN

既然考虑到了爆破，那么爆破的可行性如何？

16*255 种可能，完全可以接受。

延时逻辑已经被干掉。

本来我们也不是通过TWI输入数据，相关函数可跳过，进一步节省时间。

可以通过gdb脚本实现自动化。

既然可行，那么接下来相对就要简单了。

通过断点下套，跳过耗时的逻辑。

通过之前的copy “flag{” 和写入 “}”来绕过分支点。

自动生成测试数据，填充到flag区域。

在check判定成功、失败节点上下断点，从而获取次测试结果。

具体实现，请参照：

脚本使用：

启动模拟器

启动gdb

source help.gdb

lk

pwn

总结

本次的题目可以说非常折磨人，但是非常有趣。如果不研究到最后，不太容易想到需要爆破。如果没想到要爆破，也就不明白为啥前面有个奇怪的延时。直到最后，所有的线索都串起来了，豁然开朗。不过，虽然题目解开了，但是失去的头发和爆的肝也找不回来了。

最后，感谢出题人设计出这么有趣的题目。感谢Jarvis OJ提供了这么一个方便、高效又有趣的平台。对开帮助笔者这样的小白学习知识、开阔眼界提供了巨大的帮助。