某嵌入式设备固件升级包

发表于 2021-01-25 更新于 2021-02-16 分类于 [Challenge](#) , [2020](#) , [工业信息安全技能大赛](#) , [济南站](#)
[Challenge](#) | [2020](#) | [工业信息安全技能大赛](#) | [济南站](#) | [某嵌入式设备固件升级包](#)

[点击此处](#)获得更好的阅读体验

WriteUp来源

来自MO1N战队

题目描述

对于嵌入式设备，升级包中含有真正的固件，在漏洞挖掘之前有必要对固件进行一些处理，提高分析效率。对 UpgradePackage.bin 进行分析，列出固件符号表项数(4bytes)、bsolute_C14SequenceNumber 函数所对应的文件偏移(4bytes)。flag{符号表项数+函数文件偏移}，为十六进制8字节数据：不足四字节高位补0，例：
symbol_item_counts: 0x112233 补齐4字节: 0x00112233。

题目考点

- 固件分析
- VxWorks

解题思路

固件分析

通过Binwalk进行自动分析可以发现，NOE 771 模块的升级包NOE77101.bin 中内嵌了一个使用zlib压缩的文件。

□

通过调用Binwalk的-e参数进行自动提取后，Binwalk会把自动提取后的文件以偏移地址命名并存储在特定的目录中。

□

继续使用Binwalk对提取的385文件进行分析可以确认，该文件的确是我们所需要分析的VxWorks固件，因此提取的385文件也就是我们需要分析的固件文件了。

□

分析固件内存加载地址

为了对VxWorks系统固件进行逆向分析，我们首先必须要知道固件在内存中的加载地址。加载地址的偏差会影响到一些绝对地址的引用例如跳转函数表、字符串表的引用等。

下图是VxWorks系统在PowerPC架构下的内存分布图，如图所示VxWorks的系统固件将会被加载到一个由BSP (Board Support Package)定义的内存地址中。

□

分析固件头部的初始化代码，寻找加载地址的特征

在很多情况下我们拿到的固件是没有采用ELF格式封装的，这时就需要我们通过对固件的某些特征进行分析来判断具体的加载地址。还是以施耐德的NOE 711固件为例，在具体分析某个固件时首先我们需要知道目标设备的CPU架构，具体可以如下图所示通过binwalk -A指令来对固件的CPU架构进行分析。

□

在得知目标CPU架构后就可以使用IDA加载固件并对其代码进行初步分析。

□

□

下图是默认加载后的IDA界面，仅仅分析出了极少数的函数。接下来就需要根据固件头部的这段代码来寻找加载地址的特征。

在固件头部有如下图所示的一段有趣的代码，在对r1和r3寄存器进行赋值后进行了跳转。

指令解析 (lis:立即数载入并左移)

```
1 lis r3,0x1234
2 addi r3,r3,0x5678
```

PowerPC的每条指令都是32bit，除去指令和寄存器参数编码，只剩下16bit的长度描述立即数，如立即数加载指令li

这样立即数SIMM只有16位，所以需要两次加载，使用lis(立即数载入并左移)和addi(立即数加法)两条指令完成。

PowerPC寄存器

下图是PowerPC的寄存器用途说明，从图中可以看到R1寄存器是栈指针，而R3寄存器则是第一个参数。

现在回到我们之前看的固件头部代码处，这段代码相当于是先将栈地址设置为0x10000，将第一个参数(r3寄存器)设置为0x0，随后在栈上开辟0x10个字节的空间后跳转到当前地址+0x1cd34处执行。

根据VxWorks官网文档对内存布局的描述，Initial Stack是usrInit函数的初始化栈。

而usrInit函数则是VxWorks系统引导后运行的第一个函数，再结合之前我们分析的那段代码，可以基本确定在大部分情况下第一个跳转的地址就是usrInit这个函数的地址。

随后我们再回忆一下之前看到的VxWorks PowerPC内存布局图可以发现，初始化栈的地址同时也是固件的内存加载地址，因此r1寄存器指向的0x10000就是我们所寻找的固件加载地址。

在分析出固件加载地址后就可以使用新的加载地址重新加载固件进行分析了,通过ida修改加载基址

设置value:0x10000

利用符号表修复函数名

虽然IDA此时能够正确的识别函数及其调用关系，但依然无法自动识别出函数名，这对固件的分析工作造成了很大的阻碍。此时可以查看固件在编译时是否编入了符号表，如固件编入了符号表那么我们就可以利用符号表中的内容来修复IDA中所显示的函数名。

通过使用binwalk可以帮助我们辅助分析VxWorks固件中是否编入了符号表，并识别出符号表在固件中的位置。如下图所示binwalk识别出的符号表地址在文件偏移0x301E74处。

如下图所示，VxWorks 5系列的符号表有他独特的格式，他以16个字节为一组数据，前4个字节是0x00，之后是符号名字符串所在

的内存地址，后4个字节是符号所在的内存地址，最后4个字节是符号的类型，例如0x500为函数名。

函数名字符串内存地址: 0x0027655C

ROM: 0x0027655C aADecodeinteger::string "A_DecodeInteger"

函数内存地址: 0x001FF058

基于符号表的特征，我们能够轻松的获取到固件中符号表的起始及结束位置。

symbol_table_start = 0x301e60

symbol_table_end = 0x3293b0

此时我们就可以使用IDA的api来修复函数名，使用加载地址0x00重新加载固件后使用如下图所示的Python脚本即可进行修复。

```
1 # coding=utf-8
2 from idaapi import *
3 import time
4
5 # 符号表间隔
6 symbol_interval = 16
7
8 # 固件内存加载地址
9 load_address = 0x10000
10
11 #符号表在内存中的起始地址
12 symbol_table_start = 0x301e60 + load_address
13 # 符号表在内存中结尾地址
14 symbol_table_end = 0x3293b0 + load_address
15 # 符号表项数
16 symbol_item_counts = 0x2755
17
18 # 在IDA 中rebase 程序到加载地址
19 # 修改基地址
20 rebase_program(load_address, 0x0008)
21
22 #调用IDA的自动分析功能
23 autoWait()
24
25 ea = symbol_table_start
26 while ea < symbol_table_end:
27     offset = 4 # 每4个字节为一组数据
28
29     # 将函数名指针位置的数据转化为字符串
30     sName = GetString(Dword(ea + offset), -1, ASCSTR_C)
31     # 将函数名赋值给变量sName
32     print sName
33     if sName:
34         # 开始修复函数名
35         eaFunc = Dword(ea + offset + 4)
36         MakeName(eaFunc, sName)
37         MakeCode(eaFunc)
38         MakeFunction(eaFunc, BADADDR)
39     ea += symbol_interval
40
```

完成函数名修复后的IDA界面如下图所示，通过修复符号表IDA识别出了8000多个函数。至此VxWorks系统固件的预处理工作就全部完成了，现在我们就可以根据函数名来对一些关键服务的代码进行静态分析了。

最终得出

symbol_table_start = 0x00301e60

symbol_table_end = 0x003293b0

`symbol_item_counts = 0x00002755`

`offset_absolute_C14SequenceNumber = 0x0011fd38 - 0x10000 = 0x0010FD38`

0x0011fd38为函数内存地址，0x10000为固件加载地址，0x0010FD38为函数文件偏移

Flag

```
1 flag{000027550010FD38}
```

- 本文作者: CTFHub
- 本文链接: <https://writeup.ctfhub.com/Challenge/2020/工业信息安全技能大赛/济南站/pka8PC6FDDQC7A8Nk5yYNS.html>
- 版权声明: 本博客所有文章除特别声明外, 均采用 [BY-NC-SA](#) 许可协议。转载请注明出处!

[#Challenge #2020 #工业信息安全技能大赛 #济南站](#)
[工业物联网智能网关数据分析](#)
[dddynamic](#)