西湖论剑 easyCpp writeup

原创

Code Segment 于 2021-01-13 22:20:39 发布 43 收藏

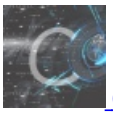
分类专栏： CTF CTF Reverse 文章标签： c++

本文链接：https://blog.csdn.net/qq_43547885/article/details/112596114

CTF 同时被 2 个专栏收录

5 篇文章 0 订阅

订阅专栏



CTF Reverse

6 篇文章 0 订阅

订阅专栏

文章目录

- 一道经典的 C++ 的逆向，做完之后总结一下

分析

- 64位elf，无壳，直接拖入 IDA 中进行分析
- 主要的逻辑全部在 `main` 函数中实现

Part01：初始化

- 首先对 `vector` 等变量进行初始化，然后进入两个 `for` 循环，先是从命令行读入 16 个数，然后调用 `fib` 函数初始化另一个 `vector`。

```
  v32 = __readfsqword(0x28u);
  std::vector<int>::vector(v24, argv, envp);
  std::vector<int>::vector(v25, argv, v3);
  std::vector<int>::vector(v26, argv, v4);
  std::vector<int>::vector(v27, argv, v5);
  std::vector<int>::vector(v28, argv, v6);
  for ( i = 0; i <= 15; ++i )
  {
    scanf("%d", &v31[i]);
    std::vector<int>::push_back(v25, &v31[i]);
  }
  for ( j = 0; j <= 15; ++j )
  {
    LODWORD(v30[0]) = fib(j);
    std::vector<int>::push_back(v24, v30);
  }
```

- `fib` 函数就是计算斐波那契数列

```
__int64 __fastcall fib(int a1)
{
  int v2; // ebx

  if ( !a1 || a1 == 1 )
    return 1LL;
  v2 = fib(a1 - 1);
  return v2 + (unsigned int)fib(a1 - 2);
}
```

Part02：调用 `transform` 对 `vector` 进行操作

- 先来整理一下 C++ STL 里面 `transform` 函数的用法，它定义在 `<algorithm>` 头文件中。**我强烈建议对于库函数直接翻到源文件看解释，没有比这个更言简意赅的了**
- 下图中展示了接收 4 个参数的 `transform` 函数的含义，其是将 `[first,last)` 间的元素按照 `op` 进行变换后，出入到 `result` 迭代器中

```
/**
```

```
  *  @brief Perform an operation on a sequence.
  *  @ingroup mutating_algorithms
  *  @param  __first     An input iterator.
  *  @param  __last      An input iterator.
  *  @param  __result    An output iterator.
  *  @param  __unary_op  A unary operator.
  *  @return   An output iterator equal to @p __result+(__last-__first).
  *
  *  Applies the operator to each element in the input range and assigns
  *  the results to successive elements of the output sequence.
  *  Evaluates @p *(__result+N)=unary_op(*(__first+N)) for each @c N in the
  *  range @p [0,__last-__first).
  *
  *  @p unary_op must not alter its argument.
 */
<_InputIterator,_OutputIterator,_UnaryOperation>
  _OutputIterator
  transform(_InputIterator __first, _InputIterator __last,
        _OutputIterator __result, _UnaryOperation __unary_op)
  {
    // concept requirements
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator>)
    __glibcxx_function_requires(_OutputIteratorConcept<_OutputIterator,
      // "the type returned by a _UnaryOperation"
      __typeof__(__unary_op(*__first))>)
    __glibcxx_requires_valid_range(__first, __last);

    for (; __first != __last; ++__first, (void)++__result)
  *__result = __unary_op(*__first);
    return __result;
  }
```

- 在源程序中，对变量重命名一波之后就会非常清晰，可以看到是将输入的第二个数字和最后一个数字之间的内容经过 `lambda` 表达式的变换输入到 `midVec` 中

```
std::vector<int>::push_back(midVec, iptFir);
midVecIns = std::back_inserter<std::vector<int>>(midVec);
iptVecEnd = std::vector<int>::end(iptVec);
iptVecBeg[0] = std::vector<int>::begin(iptVec);
iptVecSec = __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator+(iptVecBeg, 1LL);
std::transform<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::back_insert_iterator<std::vector<int>>,main::{lambda(int)#1}>(
  iptVecSec,
  iptVecEnd,
  midVecIns,
  iptFir);
```

- 关键就在于这个 `lambda` 表达式，它其实很简单，就是将传入的两个参数相加再返回

```
1 __int64 __fastcall main::{lambda(int)#1}::operator()(_DWORD **a1, int a2)
2 {
3   return (unsigned int)(**a1 + a2);
4 }
```

- 看一下上层的函数，发现传入的第一个参数始终是 `transform` 的第四个参数，传入的第二个参数是迭代过程中迭代到的值

```
v12 = __readfsqword(0x28u);
while ( (unsigned __int8)__gnu_cxx::operator!=<int *,std::vector<int>>(&v10, &v9) )
{
  value = (unsigned int *)__gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator*(&v10);
  mid = main::{lambda(int)#1}::operator()(&a4_, *value);
  v5 = std::back_insert_iterator<std::vector<int>>::operator*(&output);
```

```
  std::back_insert_iterator<std::vector<int>>::operator=(v5, &mid);
  __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator++(&v10);
  std::back_insert_iterator<std::vector<int>>::operator++(&output);
}
return output;
```

- 总结一下这部分的代码做的事：将输入的数据中第2~16个数加上第一个数，结果保存在 `midVec` 中

Part03：调用 `accumulate` 对数组元素进行处理

- 还是先看一下 `accumulate` 函数的注释
- 这个函数用于将一个 `[first,end)` 之间的数累加起来，初始值和累加的方法由参数3和参数4指定

```
/**
 *  @brief  Accumulate values in a range with operation.
 *
 *  Accumulates the values in the range [first,last) using the function
 *  object @p __binary_op.  The initial value is @p __init.  The values are
 *  processed in order.
 *
 *  @param  __first  Start of range.
 *  @param  __last  End of range.
 *  @param  __init  Starting value to add other values to.
 *  @param  __binary_op  Function object to accumulate with.
 *  @return  The final sum.
 */
<_InputIterator,_Tp,_BinaryOperation>
  inline _Tp
  accumulate(_InputIterator __first, _InputIterator __last, _Tp __init,
        _BinaryOperation __binary_op)
  {
    // concept requirements
    __glibcxx_function_requires(_InputIteratorConcept<_InputIterator>)
    __glibcxx_requires_valid_range(__first, __last);

    for (; __first != __last; ++__first)
  __init = __binary_op(__init, *__first);
    return __init;
  }
```

- 回到源程序，这里稍微有点绕，主要是没用的变量太多了。下面是 `accumulate` 的主要部分，可以看到主要还是对数组进行遍历

```
while ( (unsigned __int8)__gnu_cxx::operator!=<int *,std::vector<int>>(&begin, &end) )
{
  value = *(_DWORD *)__gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator*(&begin);
  std::vector<int>::vector(midVec2, a4);
  main::{lambda(std::vector<int>,int)#2}::operator()(midVec1, &a7, midVec2, value);
  std::vector<int>::operator=(a4, midVec1);
  std::vector<int>::~vector(midVec1);
  std::vector<int>::~vector(midVec2);
  __gnu_cxx::__normal_iterator<int *,std::vector<int>>::operator++(&begin);
}
std::vector<int>::vector(v12, a4);
return v12;
```

- 我们先不管其他的，先来看 `lambda` 表达式，还是挺绕的，但是好在代码比较少，整理一下，可以看出这实际上是将

- 我们先不管其他的，先来看 `lambda` 表达式。还是挺绕的，但是好在代码比较少，整理一下，可以看出这实际上是将 `value` 插入到了 `mid1` 之中，然后再将 `mid2` 中的值拷贝到 `mid1` 的后边。

```
mid1_1 = mid1;
v10 = a2;
mid2_1 = mid2;
value_1 = value;
v12 = __readfsqword(0x28u);
std::vector<int>::vector(mid1, a2, mid2);
std::vector<int>::push_back(mid1, &value_1);
mid1Ins = std::back_inserter<std::vector<int>>(mid1_1);
end = std::vector<int>::end(mid2_1);
begin = std::vector<int>::begin(mid2_1);
std::copy<__gnu_cxx::__normal_iterator<int *,std::vector<int>>,std::back_insert_iterator<std::vector<int>>>(
  begin,
  end,
  mid1Ins);
return mid1_1;
```

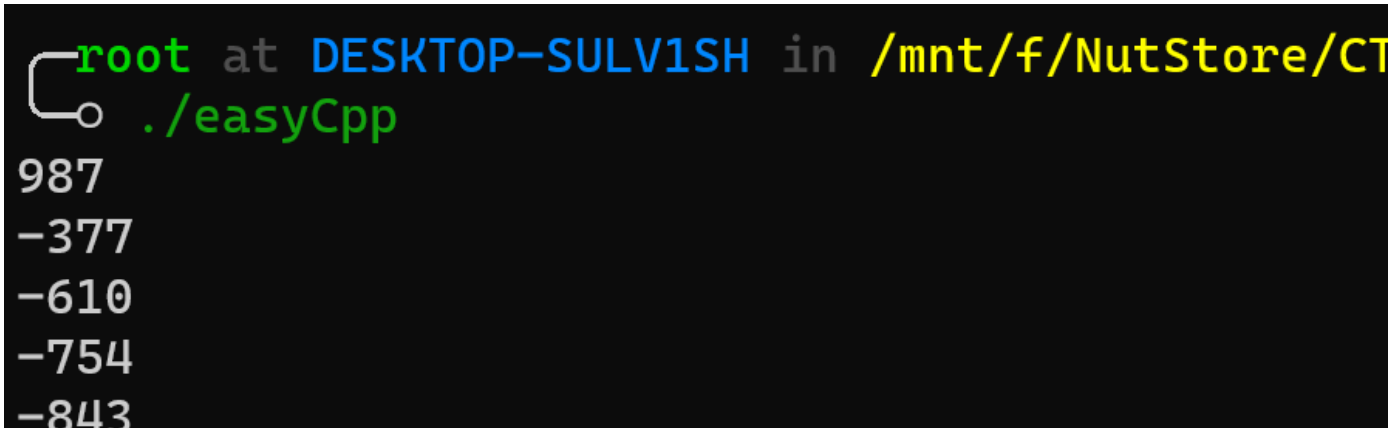
- 看懂了 `lambda` 部分，上层的就比较好懂了，最终起到的作用就是将 `[begin,end)` 之间的元素逆序的送入参数1对应的迭代器中

Part04：剩余的一点

- 先总结一下上面两部干的事：首先将输入数的2~16个分别加上第一个数，然后调用 `accumulate` 配合 `lambda` 表达式进行翻转
- 剩下的就很简单了，就是将处理过的 `iptVec` 和 `fibVec` 进行比较，若正确就输出 `flag`，很容易就能算出这 16 个数来。

```
int decrypt() {
    vector<int> fibVec;
    int cnt;
    // generate fib
    for (cnt = 0; cnt < 16; cnt++) {
        fibVec.push_back(fib(cnt));
    }
    reverse(fibVec.begin(), fibVec.end());
    for (cnt = 1; cnt < 16; cnt++) {
        fibVec[cnt] -= fibVec[0];
    }
    for (cnt = 0; cnt < 16; cnt++) {
        cout << fibVec[cnt] << endl;
    }
}
```

- 得到 flag



总结

- 我感觉这题出的真挺好的，很基础，完全没有故意把你往坑里带或者必须得靠脑洞才能过的点，全部都是 C++ 逆向的常规操作，分析通了就是秒
- 我感觉这里面涉及到 C++ 逆向的重点主要有这么几个

对于 STL 库函数的分析

- 很多函数名一看上去就懵逼，尤其是 C++ 的函数名经常有一个屏幕的宽度那么长，但其实很多函数都可能是库函数，去逆向之前不妨先在 Google 搜索一下，找找思路再开始闷头看。

lambda 表达式的逆向

- 它最误导人的地方就是 `[对象参数]` 这一部分，因为它打破了函数的封闭性，使得编译器会在生成代码的时候将上层的一些变量传到下层的栈空间中，让参数传递显得很混乱。
- 逆向过程中对于 `lambda` 表达式的分析，我觉得应该自顶向上地看，弄清楚 `lambda` 表达式中哪那部分是函数调用的参数，哪部分是对象参数，可以帮助我们分析上层的函数被调用时传入的参数。

> 推一手我的博客：buptlxr.top，求关注