Windows 自启动项的查看和分析

学号：517021910563

姓名：李昊璋

班级：F1703602

联系方式：980226547@qq.com

1.前言

1.1 目的与内容

- 目的
  - 了解 Windows 系统中可以实现自启动的技术方法，分析各自的技术原理、实现细节和隐蔽性状况
  - 编写自己的 Windows 自启动项查看软件
- 内容
  - 自启动种类必须包括：
    - Logon：启动目录，基于注册表启动
    - Services：系统服务
    - Drivers：系统驱动程序
    - Scheduled Tasks：计划任务
  - 可选研究分析内容为：
    - Internet Explorer：IE浏览器的BHO对象
    - Boot Execute：启动执行
    - Image Hijacks：映像劫
    - Known DLLs：知名动态链接库
    - Winsock Providers：Winsock服务提供程序
    - Winlogon：用户登录通知程序
    - ...

1.2 开发环境

- 集成开发环境：QT Creator 4.11.2
- 编程语言：C++
- 操作系统：windows 10 家庭版
- 编译环境：MinGW 64-bit
- 图形界面框架：QT 5.12.8

2. 自启动

2.1 基于注册表自启动

- 每次开机完成后，Windows将会查找以下注册表路径，并根据 image path 创建进程。

```
"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run"
"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Policies\\Explorer\\Run"
"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce"
"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnceEx"
"HKEY_CURRENT_USER\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run"
"HKEY_CURRENT_USER\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Policies\\Explorer\\Run"
"HKEY_CURRENT_USER\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce"
"HKEY_CURRENT_USER\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnceEx"
```

- 每当有用户登录时，Run 和 RunOnce 注册表项会让程序运行一次。子键的数据往往是一个长度不超过260字符的命令行（command line），但子键下可以写入多个入口（entries），它们的运行次序是不确定的。
- 默认情况下，在运行命令行之前会删除 RunOnce 键的值。如果 RunOnce 的值前面添加一个感叹号，则可以将值的删除推迟到命令运行之后，但没有这个前缀的情况下，假如 RunOnce 操作失败，下次启动计算机将不会要求关联的程序运行。同时，如果在安全模式下启动，RunOnce 键将被忽略，除非带有*的前缀，以强制程序在安全模式下运行。
- RunOnceEx 键只在第一次登录后被 Explorer shell 运行一次。这一注册表项的值将在处理后从注册表中删除，从此不再运行。如果 Explorer shell 不存在，就会忽略 RunOnceEx 键。不同于 Run 和 RunOnce，RunOnceEx 注册表项不会创建单独的进程（前两者的命令往往都会创建单独的进程），此外，RunOnceEx 注册表键将 entries 和 sections 按字母顺序排序，以强制性地以确定的顺序执行。
- 值得注意的是，这些键中的任何一个运行的程序都不应在其执行期间写入该键，这会干扰在该键下注册的其他程序的执行。
- HKLM 与 HKCU 分别是 HKEY_LOCAL_MACHINE 和 HKEY_CURRENT_USER 的缩写。当注册表路径含有 HKLM 时，说明该注册表表项会在计算机启动时生效。反之，路径含有 HKCU 时，是在用户登录时生效。
- 病毒可以通过 `RegSaveKey()` 和 `RegStoreKey()` 创建自启动键值。病毒首先使用 `RegSaveKey()` 将要更改的目标子键和内容复制到一个临时文件下，之后在注册表内创建一个临时子键，利用 `RegStoreKey()` 将临时文件中的内容复制到临时子键下，并添加自启动程序。之后再将此临时子键内容存储到新的临时文件下，并将新临时文件内容恢复到目标子键，从而为目标子键添加了新的自启动程序。
- 病毒可以通过捕获关机事件，在关机前写入 RunOnce 目录。正如前文所述，RunOnce 键值在启动后删除，因此管理员无法发现该自启动项。

2.2 基于启动目录自启动

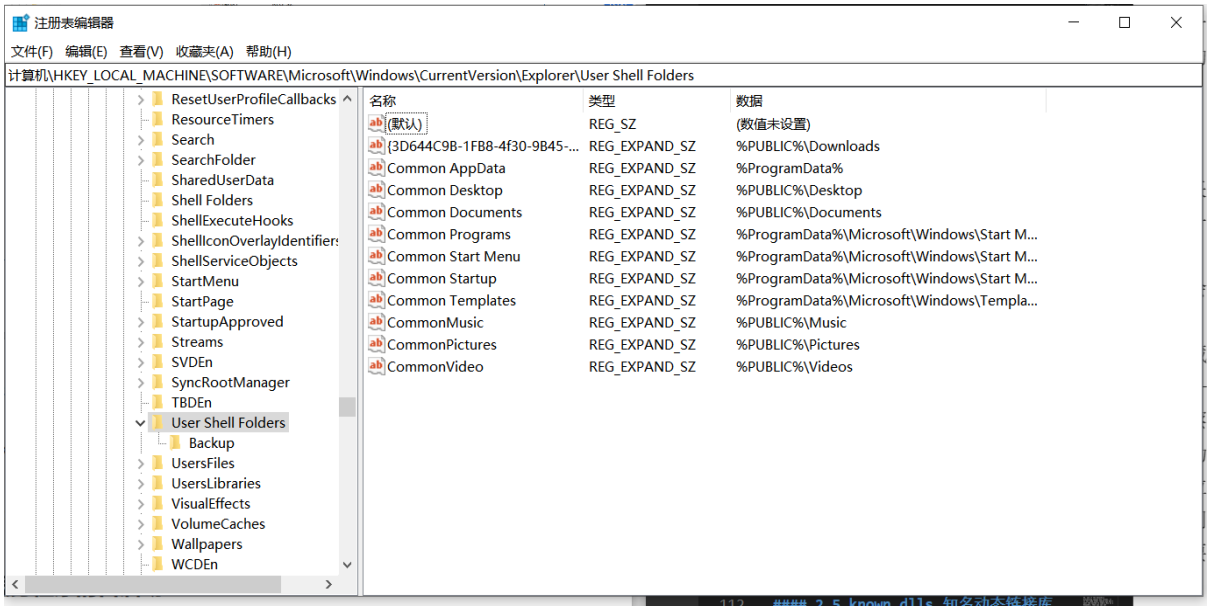
- 利用启动目录实现自启动，不需要修改系统数据，用户将所需的文件或快捷方式放入以下目录，即可在系统启动时自动加载运行，实现程序的自启动。

```
"C:\Documents and Settings\All Users\「开始」菜单\程序\启动"
"%USERPROFILE%\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup"
"C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup"
```

- 后两者目录的写入权限并不相同。前者不需要提升为管理员及以上的权限，所有用户都能够添加自启动文件，且生效时间为用户登录的时刻。后者的写入需要至少管理员的权限，因此无论哪一个用户登录，都能够自启动。

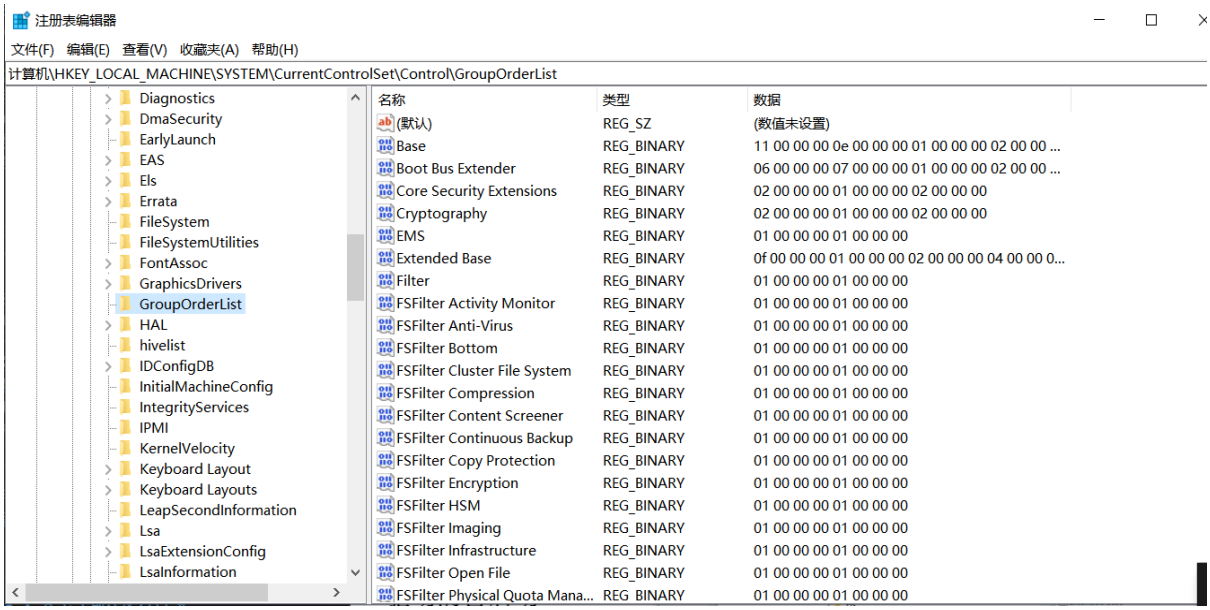
- 如果更改“启动”文件夹，可以实现隐藏的自启动。即，如果将“启动”文件夹改名为“myrun”，并新建一个名为“启动”的文件夹，系统在启动的时候，仍然会去查找“myrun”文件夹。如果将“myrun”文件夹隐藏，就能够实现隐蔽启动——但还是可以在注册表中观察到。
- 相关的注册表表目录为以下两条。可以看到，目录均位于 HKLM 下，因此启动时间为计算机启动的时刻。

```
"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders"
"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders"
```



2.3 基于系统服务和驱动的自启动

- 服务程序是后台运行的进程，常用来执行特定的任务，不需要和用户进行交互，比如自动更新服务、后台智能传输服务、事件日志服务等，而且大多数系统服务进程都是自启动进程。
- 系统通过服务管理器（SCM，Service Control Manager，对应的进程为services.exe）来管理服务，如果要创建新的服务，要求进程拥有管理员及以上的权限，并且需要额外创建服务入口函数 `ServiceMain()`，因此隐蔽性较低。
- SCM 能够自动启动服务和依赖，其中加载顺序为：ServiceGroupOrder、GroupOrderList、相关服务依赖项。



- 通过管理工具中的“服务”管理组件可查看管理用户态的系统服务（查看不到内核态的驱动服务）。
- 服务程序的配置数据位于以下注册表键值，该注册表键值存储了每一个系统服务的信息。

```
"HKLM\System\CurrentControlSet\Services"
```

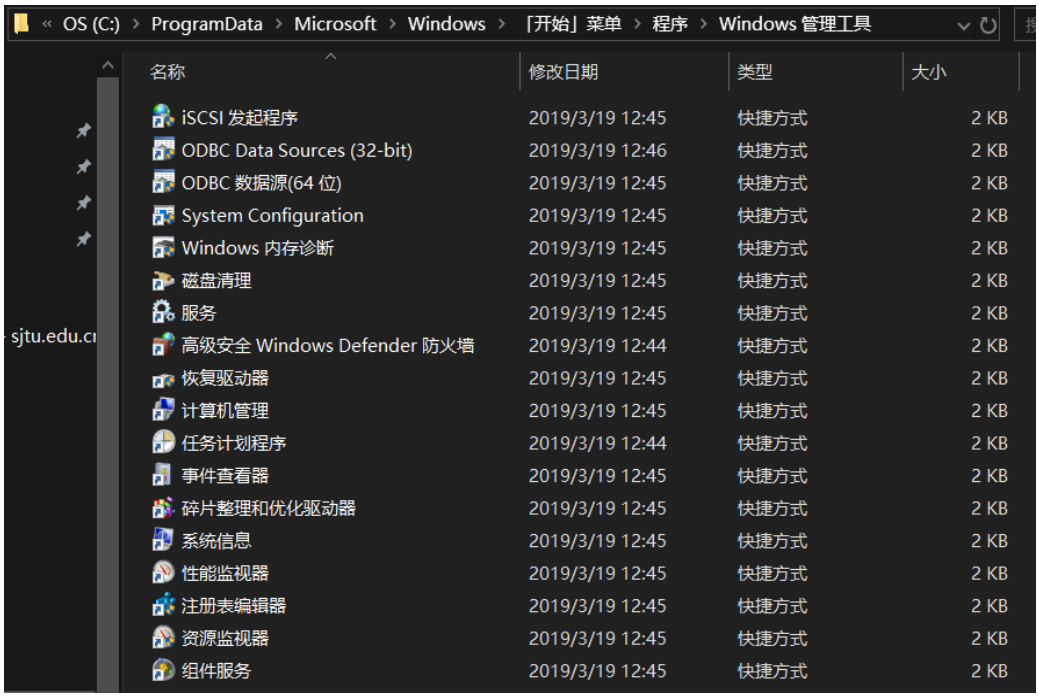
- 同时，每一个驱动都有一个形式为 `HKLM\SYSTEM\CurrentControlSet\Services\DriverName` 的键。当 PnP 管理器调用驱动程序的驱动入口例程时，管理器会传递驱动在参数 RegistryPath 中的路径。
- 注册表表项中，imagepath 保存了对应驱动程序的映像文件路径，这一路径通常为 `%SystemRoot%\system32\Drivers\DriverName.sys` ，其中 DriverName 是驱动的服务密钥（service key）。windows 通过在驱动程序的 INF 文件中使用 ServiceBinary entry 创建这个值。

2.4 基于计划任务自启动

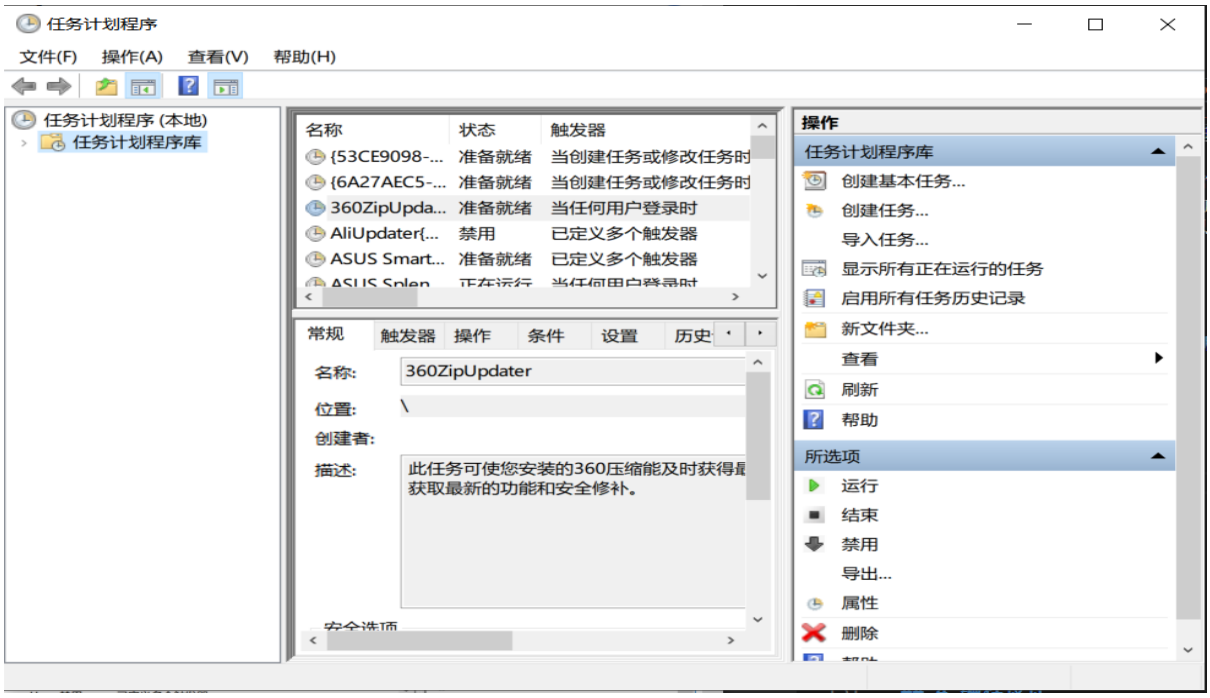
- Windows系统可以设置计划任务来执行一些定时任务，而这些任务的触发条件可以不同。任务计划程序会监控这些触发条件（时间或者事件），并在触发时执行启动指定路径程序的操作。
- 任务计划程序在每次启动操作系统时启动，它可以通过图形用户界面（GUI）或该 SDK 中描述的对应 API 运行，而编程创建新的计划任务则需要有管理员权限。
- 任务由不同的组件组成，但是任务都必须包含任务计划程序用来启动任务的触发器（trigger），和描述任务计划程序将执行的工作的动作（action）。其中，任务可以指定一个或多个触发器，也可以指定一个或多个操作来完成其工作。
- 触发器是一组条件，一项任务可以由一个或多个触发器启动，最多可以指定48个触发器。触发器分为基于时间的和基于事件的触发，当然二者可以通过组合，形成更为复杂的触发器。基于时间的触发器在指定时间启动任务，包括在特定时间启动一次任务，或按每天、每周、每月甚至每月的星期几计划多次启动。基于事件的触发器可响应某些系统事件来启动任务，例如设置为在系统启动、用户登录到本地计算机或系统变得空闲时启动等。
- 一般可以通过下面两种方式，查看任务计划程序。
  - 该程序的快捷方式在

    `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Administrative Tools\Task Scheduler.lnk`

    下。



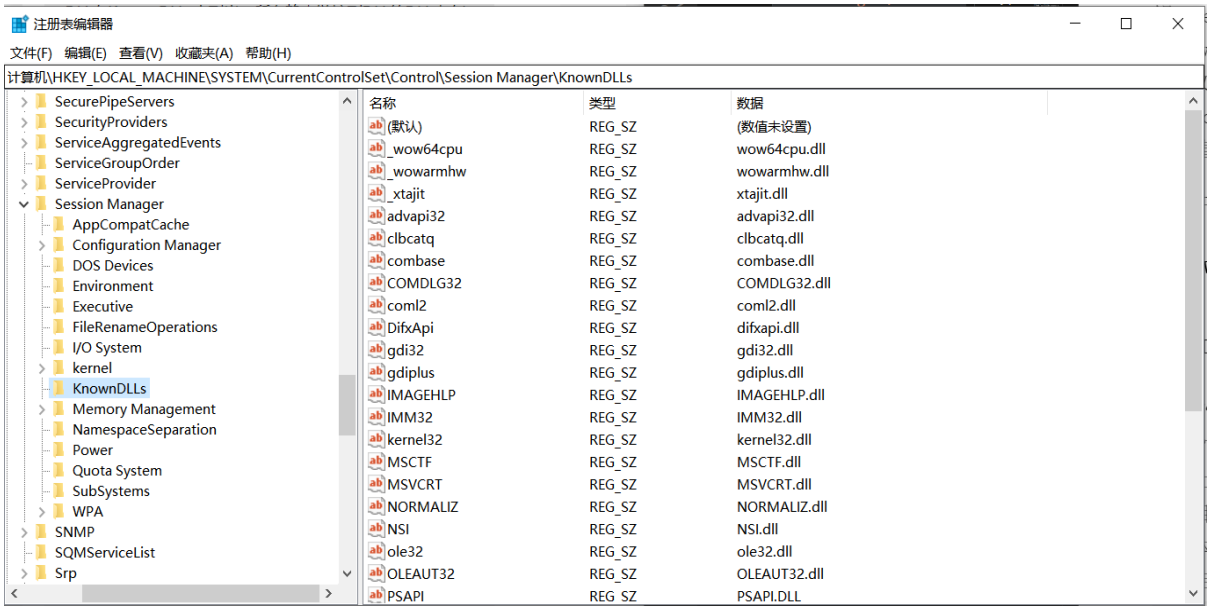
- 可以通过在cmd输入命令 `%windir%\system32\taskschd.msc /s` ，查看任务计划程序。



2.5 基于知名动态链接库自启动

- KnownDll 是 Windows NT 和 win9x 中的一种机制，允许系统“缓存”常用的系统 dll。它是一种安全机制，可以阻止木马利用弱删除应用程序目录权限改变系统 dll 的版本。
- 系统启动时，系统将会查找 `HKLM\system\CurrentControlSet\Control\Session Manager\KnownDLLs` ，并为注册表表项列出的每一个 dll 创建一个 `\KnownDlls\<dll filename>` 的 section。
- 这一注册表表项中，没有为 KnownDLLs 列出路径，因为所有的 known dll 都默认在 `HKLM\System\CCS\Control\KnownDLLs\DllDirectory` 中指定的目录中——这也一定程度上体现了 KnownDlls 作为安全机制的一个特点，通过要求 KnownDlls 在同一个目录下，攻击者更难注入他们的 KnowDll 形式的木马。

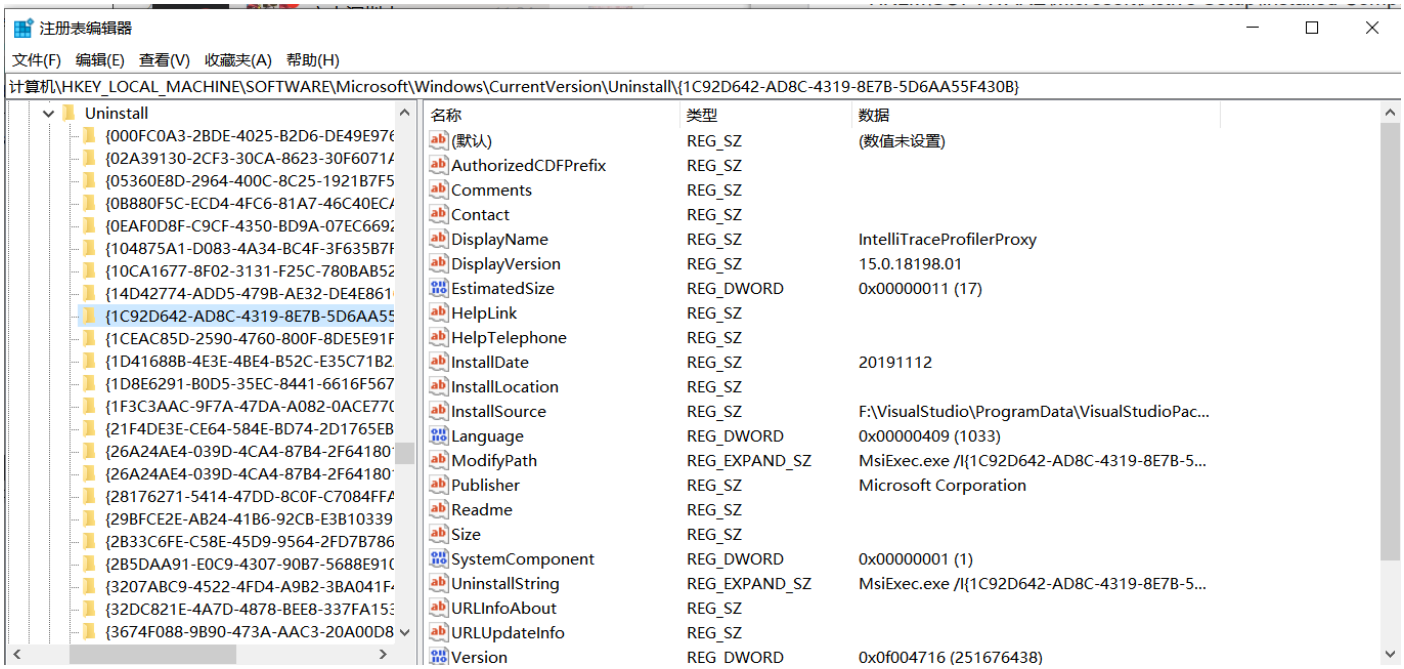
- 如果不希望系统加载一个 knowndll，则可以在 `HKLM\ system \CCS\Control\Session Manager\ExcludeFromKnownDlls` 进行设置，以 KnownDlls 处理过程中对应的 dll。
- 同样的，由于处于 `HKLM` 的目录下，此类自启动在计算机启动时就会运行，同时需要管理员及以上的权限才能够写入。



2.6 基于ActiveX控件自启动

- ActiveX 控件是 Internet Explorer 的一类小程序，通常称为附加程序。此类程序可以监视用户个人浏览习惯、安装恶意软件、生成弹出窗口、记录击键和密码以及执行其他恶意操作。值得注意的是，只需要修改 ActiveX 自启动的注册表表项即可实现自启动，而需要 ActiveX 控件的网站只有 Internet Explorer 的网站。因此，ActiveX 插件是木马病毒的良好传染途径。
- ActiveX 控件实际上并非只能作为 Internet Explorer 的附加程序，它们还可以在其他 Microsoft 应用程序（例如 Microsoft Office）中工作。
- ActiveX 控件的注册表目录为 `HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components\{XXXXXXXX-XXXX-XXXX□XXXX-XXXXXXXXXXXX}`，这一格式称为微软的GUID（Globally Unique Identifier），是一种二进制长度为128位的数字标识符。理想情况下，任何计算机和计算机集群都不会生成两个相同的GUID，因为 GUID 的总数达到了2^128个。
- 可以用 GUID 在注册表中标识软件，相关的标识关系可以通过以下三个注册表表项查看。

```
"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall"
"HKEY_CLASSES_ROOT\Installer\Products"
"HKEY_CURRENT_USER\Software\Microsoft\Installer\Products"
```



3.主要API说明

3.1 Windows的API

3.1.1 CryptQueryObject()

- 语法：

```
BOOL CryptQueryObject(
    DWORD      dwObjectType, //指定查询对象的类型
    const void *pvObject, //指向要查询的对象的指针
    DWORD      dwExpectedContentTypeFlags, //指明期望的内容类型
    DWORD      dwExpectedFormatTypeFlags, //返回类型的预期格式
    DWORD      dwFlags, //必须设置为零
    DWORD      *pdwMsgAndCertEncodingType, //消息中使用的编码类型，可设置为NULL
    DWORD      *pdwContentType, //内容的实际类型
    DWORD      *pdwFormatType, //内容的实际格式类型
    HCERTSTORE *phCertStore, //证书存储的句柄
    HCRYPTMSG  *phMsg, //打开的消息的句柄
    const void **ppvContext //有关该对象的其他信息
);
```

- 函数检索有关一个加密 API 对象的内容,例如如证书、证书吊销列表、证书信任列表。如果运行成功，函数将返回非零值，否则返回零。如果需要查看扩展的错误信息，可以调用 `GetLastError()` 。

3.1.2 CryptMsgGetParam()

- 语法

```
BOOL CryptMsgGetParam(
    HCRYPTMSG hCryptMsg, //加密消息的句柄
    DWORD     dwParamType, //要检索的数据的参数类型，这决定了pvData使用的结构类型
    DWORD     dwIndex, //检索到的参数的索引
    void      *pvData, //指向缓冲区的指针，该缓冲区接收检索到的数据
    DWORD     *pcbData //指向变量的指针，该变量指定pvData参数指向的缓冲区的大小（以字节为单位）
);
```

- 对加密消息进行编码或解码之后，该函数可以获取消息参数。同样的，函数调用成功将返回非零值，否则返回零。

3.1.3 CertFindCertificateInStore()

- 语法

```
PCCERT_CONTEXT CertFindCertificateInStore(
    HCERTSTORE     hCertStore, //要搜索的certificate store的句柄
    DWORD          dwCertEncodingType, //指定使用的编码类型
    DWORD          dwFindFlags, //常设置为0，用于修改搜索条件
    DWORD          dwFindType, //进行搜索的类型。搜索类型确定数据类型与内容
    const void     *pvFindPara, //指向dwFindType共同使用的结构与数据
    PCCERT_CONTEXT pPrevCertContext //指向函数返回的最后一个结构体
);
```

- 函数能够在证书存储区（certificate store）找到第一个或下一个证书内容，同时，该函数可以用于一个循环，以在证书存储区中查找所有匹配条件的证书。如果调用成功，函数返回一个指向只读 CERT_CONTEXT 结构的指针，否则返回 NULL。

3.1.4 CryptDecodeObject()

- 语法

```
BOOL CryptDecodeObject(
    DWORD      dwCertEncodingType, //使用的编码类型
    LPCSTR     lpszStructType, //指向定义结构类型的OID的指针
    const BYTE *pbEncoded, //要解码的编码结构的指针
    DWORD      cbEncoded, //pbEncoded指向的字节数
    DWORD      dwFlags, //启用其他功能
    void       *pvStructInfo, //指向接收已解码结构的缓冲区
    DWORD      *pcbStructInfo //指定pvStructInfo参数指向的缓冲区的大小（以字节为单位）
);
```

- 函数解码参数 lpszStructType 所确定的类型结构。如果调用成功，则返回值为非零，否则返回零。

3.1.5 WINTRUST_DATA

- 语法

```
typedef struct _WINTRUST_DATA {
    DWORD                                   cbStruct; //此结构的大小（以字节为单位）
    LPVOID                                  pPolicyCallbackData;
    LPVOID                                  pSIPClientData; //指向数据缓冲区的指针
    DWORD                                   dwUIChoice; //指定要使用的用户界面（UI）的类型
    DWORD                                   fdwRevocationChecks; //证书吊销检查选项
    DWORD                                   dwUnionChoice; //指定要使用的联合成员，并因此指定要为其验证
    union {
    #if ...
        WINTRUST_FILE_INFO_           *pFile; //指向WINTRUST_FILE_INFO结构的指针
    #else
        struct WINTRUST_FILE_INFO_    *pFile;
    #endif
    #if ...
        WINTRUST_CATALOG_INFO_        *pCatalog; //指向WINTRUST_CATALOG_INFO结构的指针
    #else
        struct WINTRUST_CATALOG_INFO_ *pCatalog;
    #endif
    #if ...
        WINTRUST_BLOB_INFO_           *pBlob; //指向WINTRUST_BLOB_INFO结构的指针
    #else
        struct WINTRUST_BLOB_INFO_    *pBlob;
    #endif
    #if ...
        WINTRUST_SGNR_INFO_           *pSgnr; //指向WINTRUST_SGNR_INFO结构的指针
    #else
        struct WINTRUST_SGNR_INFO_    *pSgnr;
    #endif
    #if ...
        WINTRUST_CERT_INFO_           *pCert; //指向WINTRUST_CERT_INFO结构的指针
    #else
        struct WINTRUST_CERT_INFO_    *pCert;
    #endif
    };
    DWORD                                   dwStateAction; //指定要采取的措施
    HANDLE                                  hWVTStateData; //状态数据的句柄
    WCHAR                                   *pwszURLReference; //保留
    DWORD                                   dwProvFlags;
    DWORD                                   dwUIContext;
    struct WINTRUST_SIGNATURE_SETTINGS_ *pSignatureSettings;
} WINTRUST_DATA, *PWINTRUST_DATA;
```

- 当调用 `WinVerifyTrust()` 向 `trust providers` 传递必要信息时需要用到这个结构体。

3.1.6 WinVerifyTrust()

- 语法

```
LONG WinVerifyTrust(
    HWND   hwnd, //调用者的窗口可选句柄。用此确定是否可与用户交互
    GUID   *pgActionID, //指向 GUID 结构体
    LPVOID pWVTData //可转换为 WINTRUST_DATA结构的指针
);
```

- 函数执行指定对象的信任验证操作。除零外，没有其他值可以视为函数成功返回。

3.1.7 VerQueryValueA()

- 语法

```
BOOL VerQueryValueA(
    LPCVOID pBlock, //GetFileVersionInfo())返回的版本信息资源
    LPCSTR  lpSubBlock, //要获取的版本信息值
    LPVOID  *lplpBuffer, //存储版本信息
    PUINT   puLen //lplpBuffer对应缓冲区大小
);
```

- 从指定的版本信息资源中检索指定的版本信息。在调用之前，必须先调用 `GetFileVersionInfoSize()` 和 `GetFileVersionInfo()` 。如果指定的版本信息结构存在，并且版本信息可用，则返回值为非零，反之，如果指定的名称不存在或指定的资源无效，则返回值为零。

3.1.8 GetFileVersionInfoSize()

- 语法

```
DWORD GetFileVersionInfoSizeA(
    LPCSTR  lptstrFilename, //指定文件的名称
    LPDWORD lpdwHandle //指向该函数设置为零的变量的指针
);
```

- 函数用于确定操作系统是否可以检索指定文件的版本信息。如果版本信息可用，则返回该信息的大小（以字节为单位）。

3.1.9 GetFileVersionInfo()

- 语法

```
BOOL GetFileVersionInfoA(
    LPCSTR lptstrFilename, //文件名
    DWORD  dwHandle, //该参数被忽略
    DWORD  dwLen, //lpData参数指向的缓冲区的大小（以字节为单位）
    LPVOID lpData //指向接收文件版本信息的缓冲区
);
```

- 函数用于检索指定文件的版本信息。如果函数调用成功，则返回值为非零，否则为零。

3.1.10 CoInitializeEx()

- 语法

```
HRESULT CoInitializeEx(
    LPVOID pvReserved, //保留参数，必须为NULL
    DWORD  dwCoInit //线程的并发模型和初始化选项
);
```

- 函数用于初始化供调用线程使用的COM库，设置线程的并发模型，并在需要时为该线程创建一个新的单元。可以返回标准返回值 E_INVALIDARG，E_OUTOFMEMORY 和 E_UNEXPECTED，以及 S_OK、S_FALSE、RPC_E_CHANGED_MODE。需要说明的是，对于使用 COM 库的每个线程，必须至少调用一次该函数，并且通常只能调用一次。

3.1.11 CoInitializeSecurity()

- 语法

```
HRESULT CoInitializeSecurity(
    PSECURITY_DESCRIPTOR        pSecDesc, //服务器将用于接收呼叫的访问权限
    LONG                        cAuthSvc, //asAuthSvc参数中的条目数
    SOLE_AUTHENTICATION_SERVICE *asAuthSvc, //服务器愿意用来接收呼叫的一组身份验证服务
    void                        *pReserved1, //保留参数，必须为NULL
    DWORD                       dwAuthnLevel, //进程的默认身份验证级别
    DWORD                       dwImpLevel, //代理的默认模拟级别。
    void                        *pAuthList, //指向 SOLE_AUTHENTICATION_LIST 数组的指针指示客户端可
    DWORD                       dwCapabilities, //指示客户端或服务器的其他功能
    void                        *pReserved3 //保留参数，必须为NULL
);
```

- 函数用于注册安全性并设置该过程的默认安全性值。可以返回标准返回值 E_INVALIDARG 以及 S_OK（成功）。

3.1.12 CoCreateInstance()

- 语法

```
HRESULT CoCreateInstance(
    REFCLSID  rclsid, //关联的CLSID
    LPUNKNOWN pUnkOuter, //如果为NULL，则表示该对象不是作为聚合的一部分创建的。如果为非NULL，则指向聚
    DWORD     dwClsContext, //新创建对象的代码的运行上下文
    REFIID    riid, //与对象通信的接口标识符的引用
    LPVOID    *ppv //接收 riid 中请求的接口指针的指针变量的地址
);
```

- 函数用于创建与指定CLSID关联的类的单个对象（该对象未初始化）。函数返回 S_OK 表明已成功创建指定对象类的实例。

3.1.13 ITaskFolder

- 该接口提供用于在文件夹中注册（创建）任务，从文件夹中删除任务以及从文件夹中创建或删除子文件夹的方法。
- 主要的方法为：
  - `ITaskFolder::CreateFolder`： 创建用于相关任务的文件夹。
  - `ITaskFolder::DeleteTask`： 从文件夹中删除任务。
  - `ITaskFolder::get_Path`： 获取到文件夹存储路径。
  - `ITaskFolder::GetFolders`： 获取文件夹中的所有子文件夹。
  - `ITaskFolder::get_Name`： 获取用于标识包含任务的文件夹的名称。

- `ITaskFolder::GetTasks`： 获取文件夹中的所有任务。

3.1.14 IRegisteredTask

- 该接口功能较多，包括提供用于立即运行任务的方法，获取任务的所有正在运行实例的方法，获取或设置用于注册任务的凭据以及描述任务的属性的方法。
- 主要的方法为：
  - `IRegisteredTask::get_Definition`： 获取任务的定义。
  - `IRegisteredTask::get_LastRunTime`： 获取上一次运行已注册任务的时间。
  - `IRegisteredTask::get_Name`： 获取已注册任务的名称。
  - `IRegisteredTask::get_Path`： 获取到已注册任务的存储路径。
  - `IRegisteredTask::get_State`： 获取已注册任务的操作状态。
  - `IRegisteredTask::get_Xml`： 获取已注册任务的XML格式的注册信息。
  - `IRegisteredTask::GetRunTimes`： 获取计划在指定时间内运行已注册任务的时间。

3.1.15 ITaskService

- 该接口提供对任务计划程序服务的访问权限，以管理已注册的任务。
- 主要的方法为：
  - `ITaskService::Connect`： 连接到远程计算机，并将此接口上的所有后续呼叫与远程会话相关联。
  - `ITaskService::GetRunningTasks`： 获取正在运行的任务的集合。
  - `ITaskService::GetFolder`： 获取已注册任务的文件夹。
  - `ITaskService::NewTask`： 返回一个空的任务定义对象，将其填充设置和属性，然后使用 `ITaskFolder::RegisterTaskDefinition` 方法进行注册。

3.1.16 ITaskFolderCollection

- 该接口为包含任务的文件夹集合提供信息和控制。
- 主要的方法为：
  - `ITaskFolderCollection::get__NewEnum`： 获取文件夹集合的集合枚举器（collection enumerator）。
  - `ITaskFolderCollection::get_Count`： 获取集合中的文件夹数。
  - `ITaskFolderCollection::get_Item`： 从集合中获取指定的文件夹。

3.2 QT中的API

3.2.1 QSettings.childKeys()

- 语法

```
QStringList QSettings :: childKeys () const
```

- 返回可以使用 QSettings 对象读取的所有 top-level keys 的列表。

3.2.2 QSettings.childGroups()

- 语法

```
QStringList QSettings :: childGroups () const
```

- 返回一个包含所有键顶级组（all key top-level groups）列表，列表中的组包含可以使用 QSettings 对象读取的键。

3.2.3 QFileInfo

- 语法：

```
#include <QFileInfo>
```

- QFileInfo 类提供与系统无关的文件信息。代码中用到了函数 `QDateTime QFileInfo :: lastModified()`，用于获得目标文件的最近修改时间。

3.2.4 QFileIconProvider

- 语法：

```
#include <QFileIconProvider>
```

- QFileIconProvider 类为 QDirModel 和 QFileSystemModel 类提供了文件图标。代码中用到了函数 `QIcon icon(IconType type) const` 以获取 IconType 图标类型对应的图标

4. 编程设计

4.1 程序框架

- 头文件说明
  - `description.h`： 读取文件的描述。
  - `mainwindow.h`： 定义窗口与槽函数。
  - `publisher.h`： 验证签名（获得publisher），实现 QString 与 LPCTSTR 的类型转换。
  - `verified.h`： 验证签名（signed 与 unsigned）。
- 源文件说明
  - `description.cpp`： 读取文件的描述。
  - `main.cpp`： 程序入口。
  - `mainwindow.cpp`： 填充 UI 界面内容，实现 UI 界面中的逻辑，所有的注册表读取、签名验证等均在此文件中调用。
  - `mainwindow.ui`： 构建 UI 界面。
  - `publisher.cpp`： 验证签名的具体实现。
  - `verified.cpp`： 验证签名的具体实现。
  - `schetasks.cpp`： 读取计划任务。QT 中没有读取计划任务的接口，因此使用了 windows 的 api 进行读取。但是由于 QT Creator 的编译器问题，taskschd.lib 文件始终无法读取，编译时报

错，读取方法改为调用基于 windows api 的可执行文件，读取控制台输出结果，从而获得计划任务的名称与路径。

- 调用可执行文件说明
  - `schetasks.exe`：具体功能如上所述，将任务的名称与 image path 输出到控制台。

4.2 主要数据结构与函数说明

```
//UI 界面中 button 被点击时触发，用于切换表格，并改变 button 的底色。
void MainWindow::on_Drivers_clicked();
void MainWindow::on_Logon_clicked();
void MainWindow::on_Knowdlls_clicked();
void MainWindow::on_Service_clicked();
void MainWindow::on_ScheTasks_clicked();

//初始化 UI 界面，规定各个列的宽度，并将所有 table 隐藏起来。
void init_ui(QTableWidget *table[], int size);

//数据预处理。将值送入，替换系统变量为绝对路径，
//并删除与路径无关的内容，输出标准化的结果 image path。
QString preprocess(QString value);

//调用 `description.cpp`、`publisher.cpp`、`verified.cpp` 中的函数，
//以验证 value 的签名—包括 publisher 和 sign—并获得对应文件的描述。
//这里参数 value 已经经过预处理，为绝对路径。将以上内容填写到表格对应的位置。
void publisher_preprocess(QTableWidget *table, QString value, int row_counter);

//获得显示 logon 的表格。函数首先获得对应注册表项的所有子键，之后对每一个子键，
//获得值之后标准化为绝对路径，并由绝对路径获得时间戳与图标。
//最后通过函数 publisher_preprocess 获得文件的描述与签名验证结果。
void show_logon_table(QTableWidget *table, QString regedit_path);

//过程类似 `show_logon_table()`，但获取的是子目录而不是子键。
//由于 service 和 drivers 位于同一注册表目录下，因此需要根据值的内容判断是否为 service。
void show_services_table(QTableWidget *table_services, QString regedit_path);

//该函数与 `show_services_table` 基本一致，区别仅为前者的值不包含 `system32\drivers\`，后者的值包含。
void show_drivers_table(QTableWidget *table_drivers, QString regedit_path);

//该函数首先调用可执行程序`schetasks.exe`，以获得计划任务的所有内容与执行路径。
//之后处理可执行程序的输出结果，并获得签名验证信息和描述，填入 table 对应位置。
void show_schetasks_table(QTableWidget *table_schetasks);

//函数逻辑与 `show_logon_table()` 几乎一致。
//但由于注册表路径下，KnownDlls 的值不为绝对路径，因此获取值之后需要添加前缀路径。
//正如前文所述，KnownDlls 均在同一目录下，因此只需在值之前添加相同路径信息，即可获得绝对路径。
//后面操作一致：获得图标、时间戳、签名等信息后填入 table。
void show_knowndlls_table(QTableWidget *table_knowndlls, QString regedit_path);
```

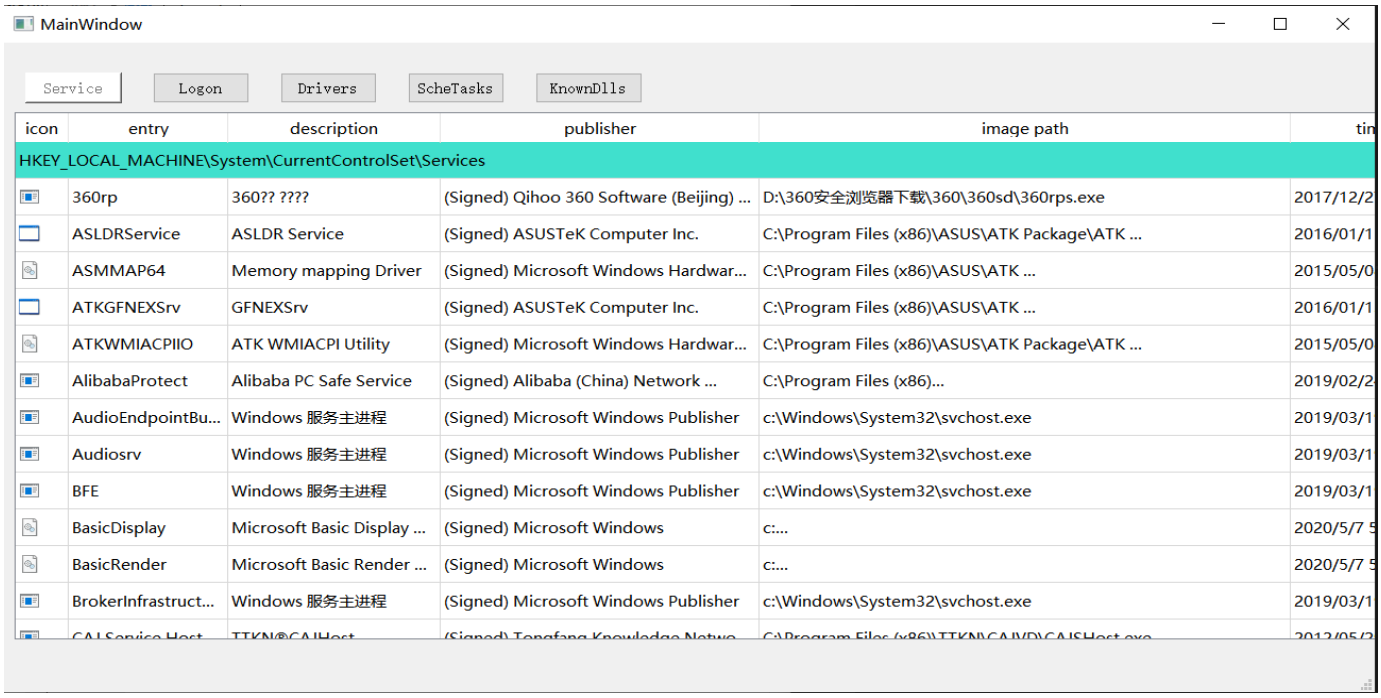
```
//由绝对路径获得文件的描述，并通过指针获得输出结果。
BOOL get_file_version_string(LPCTSTR pFileName, LPCTSTR pName, LPTSTR ptBuf, UINT lenBuf);

//由绝对路径验证签名，调用的函数通过参数中的 QString 指针获得输出结果。
BOOL VerifyEmbeddedSignature(LPCWSTR pwszSourceFile, QString *sign)

//获得 Publisher 的相关信息，之后传给上层调用函数，上层函数通过 QString 指针获得 publisher 信息
BOOL GetProgAndPublisherInfo(PCMSG_SIGNER_INFO pSignerInfo, PSPROG_PUBLISHERINFO Info);

//递归搜索子文件夹。获取目录下的 task 与各个文件夹。
void get_folders(ITaskFolder* root_folder, HRESULT hr);

//获得文件夹下已注册的任务
void get_tasks(ITaskFolder* folder, HRESULT hr);
```

5. 运行效果

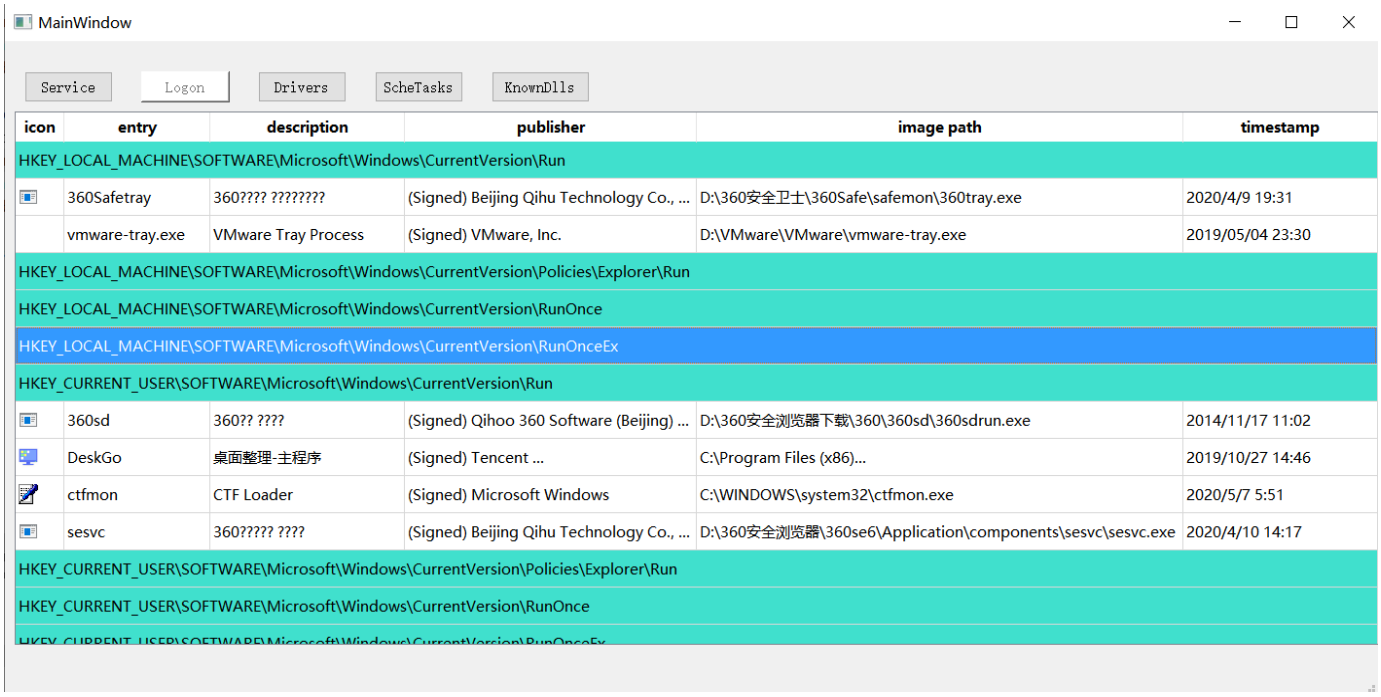
5.1 运行效果展示

- 初始界面

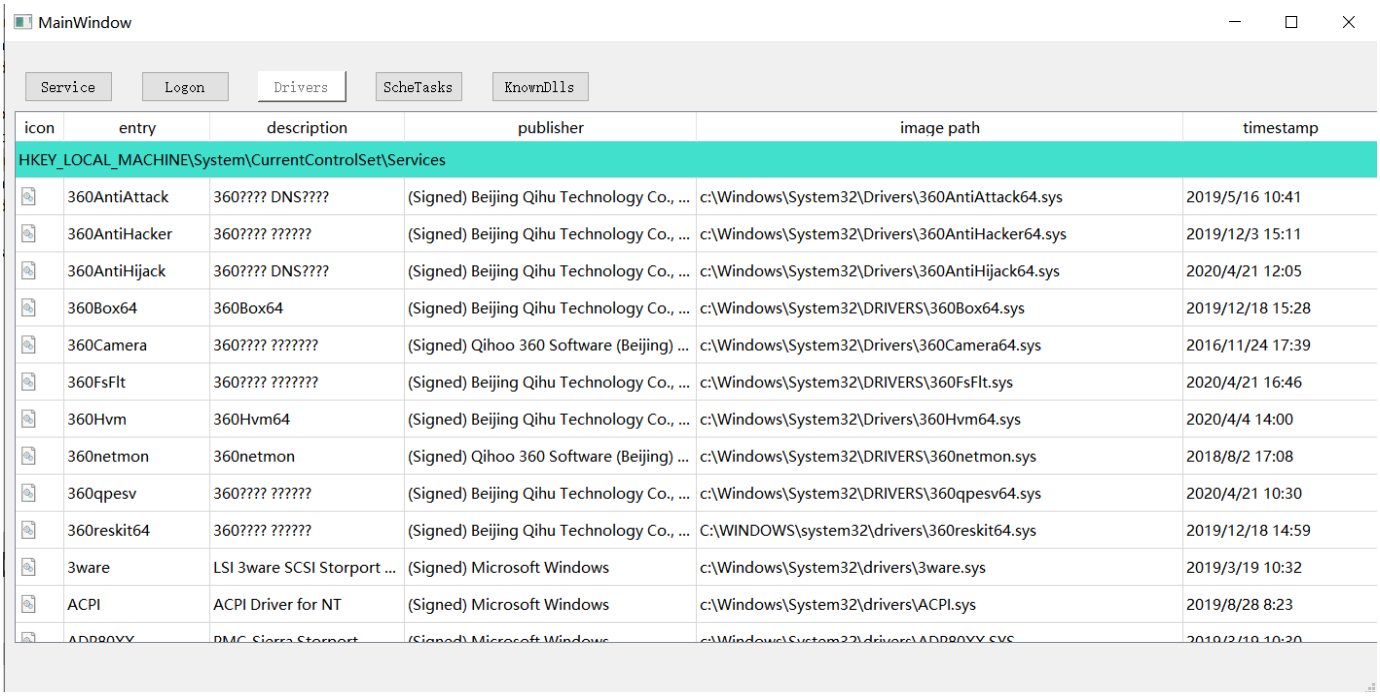
- Services



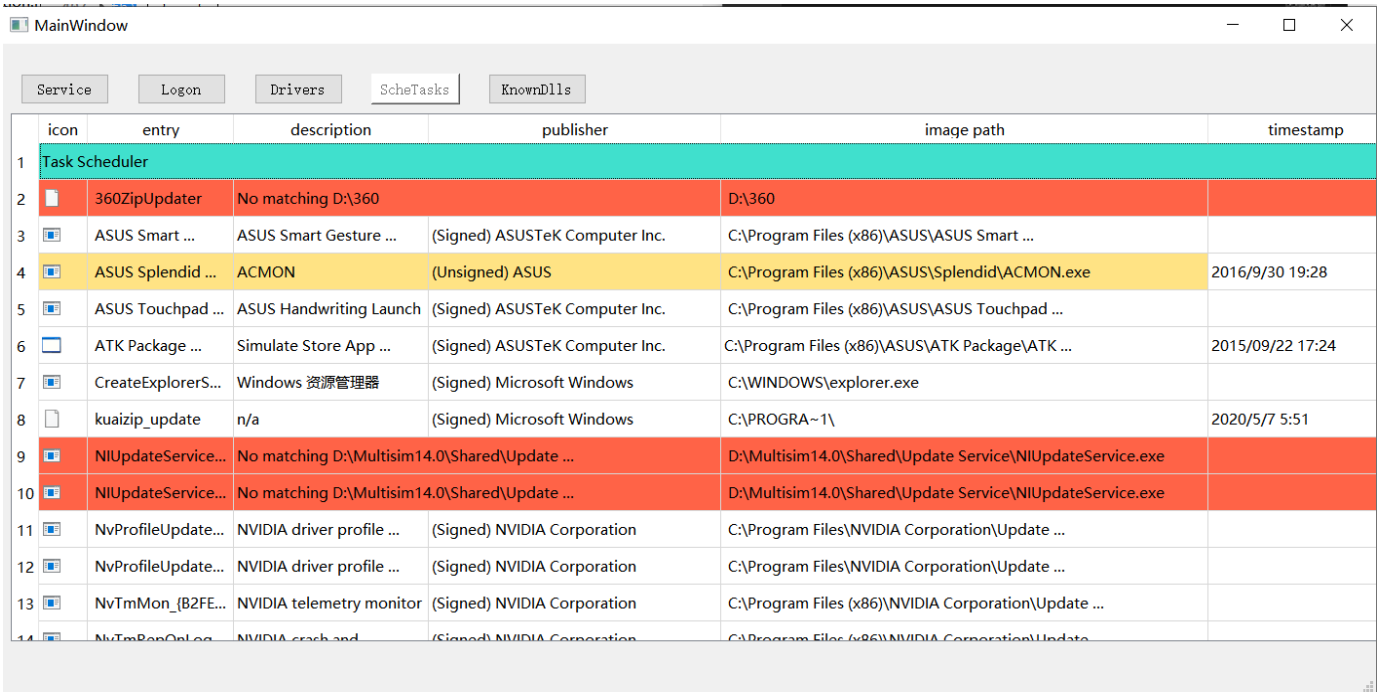
- Logon



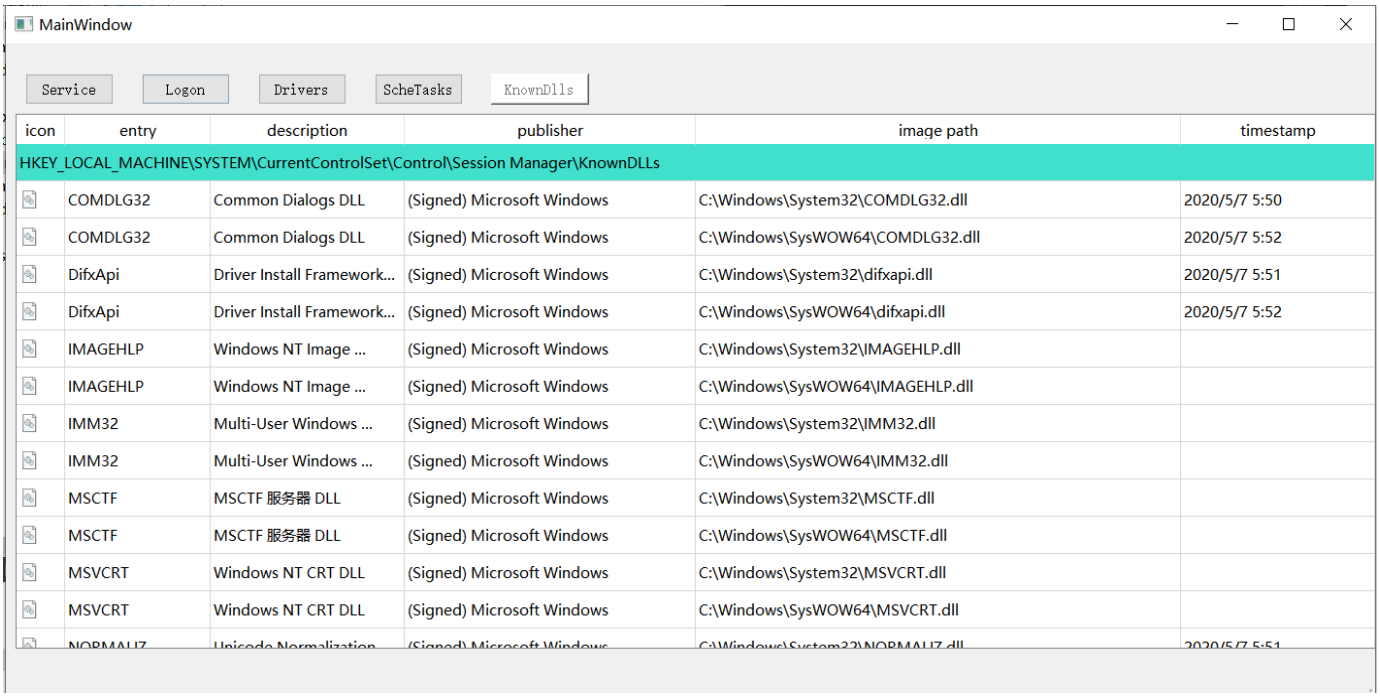
- Drivers



- Schedule Tasks

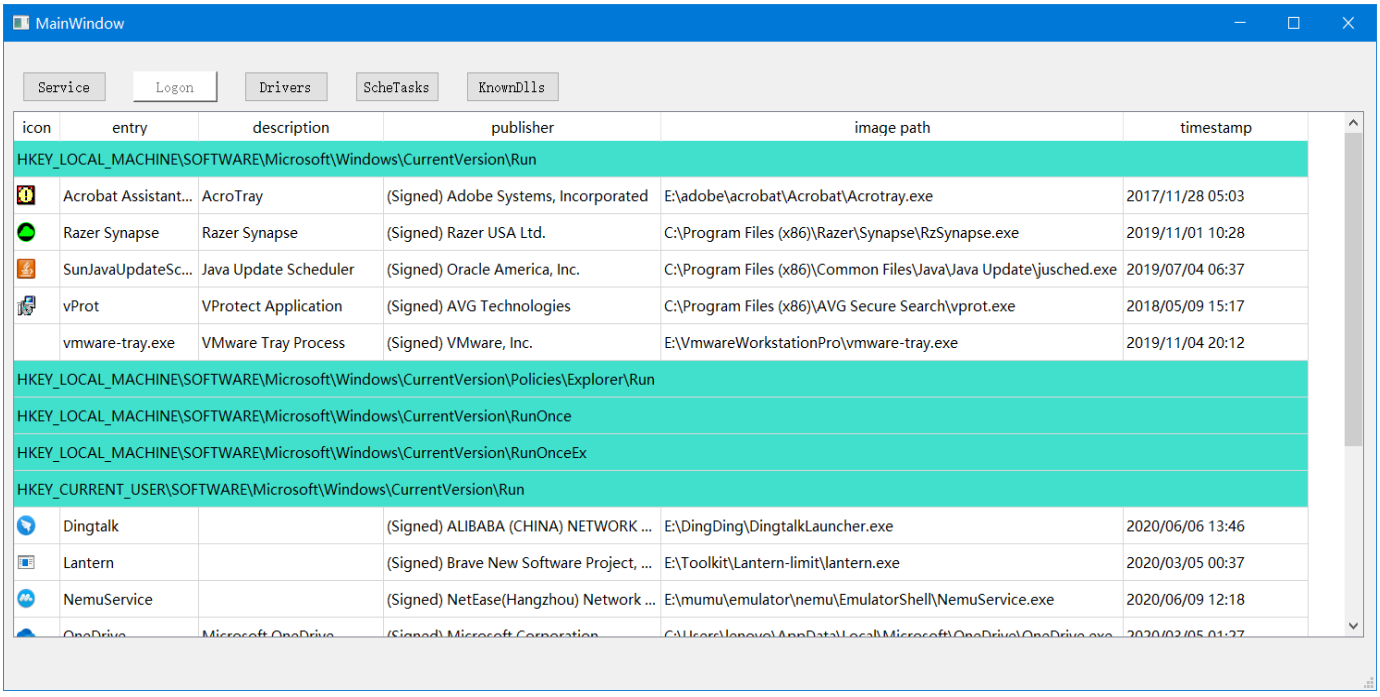


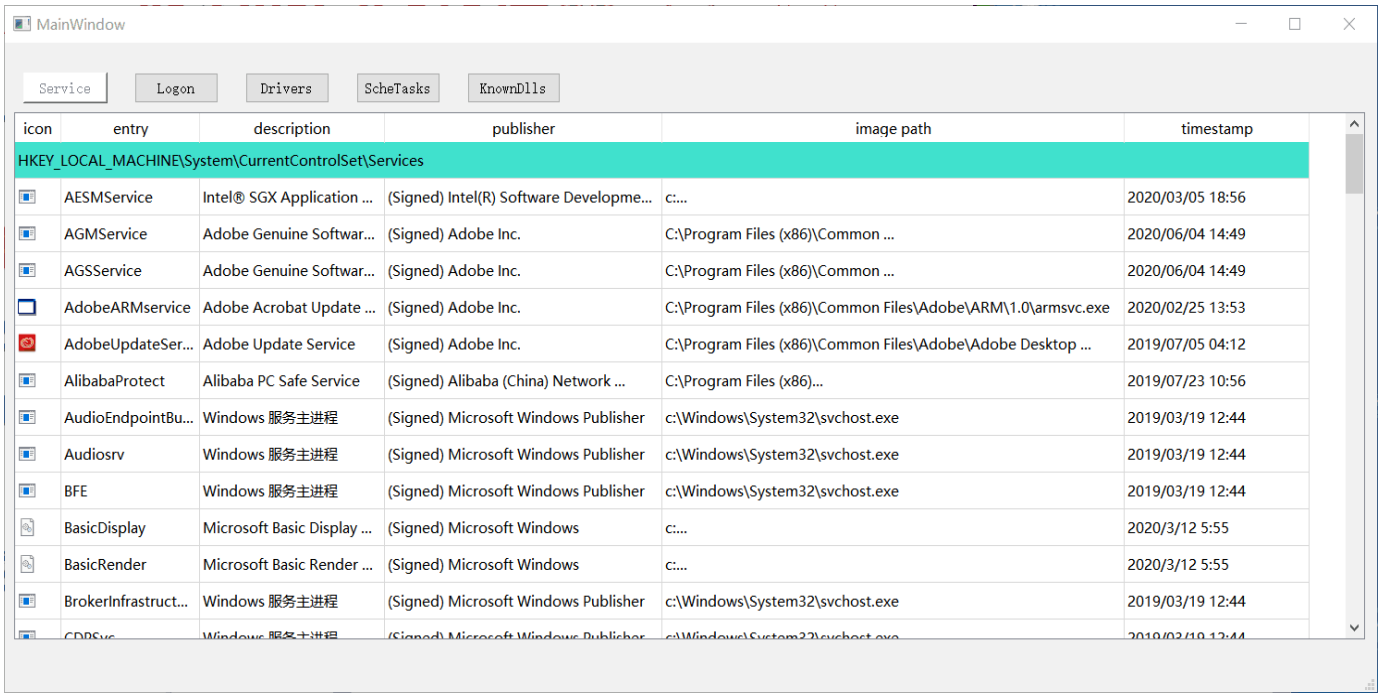
- Known Dlls

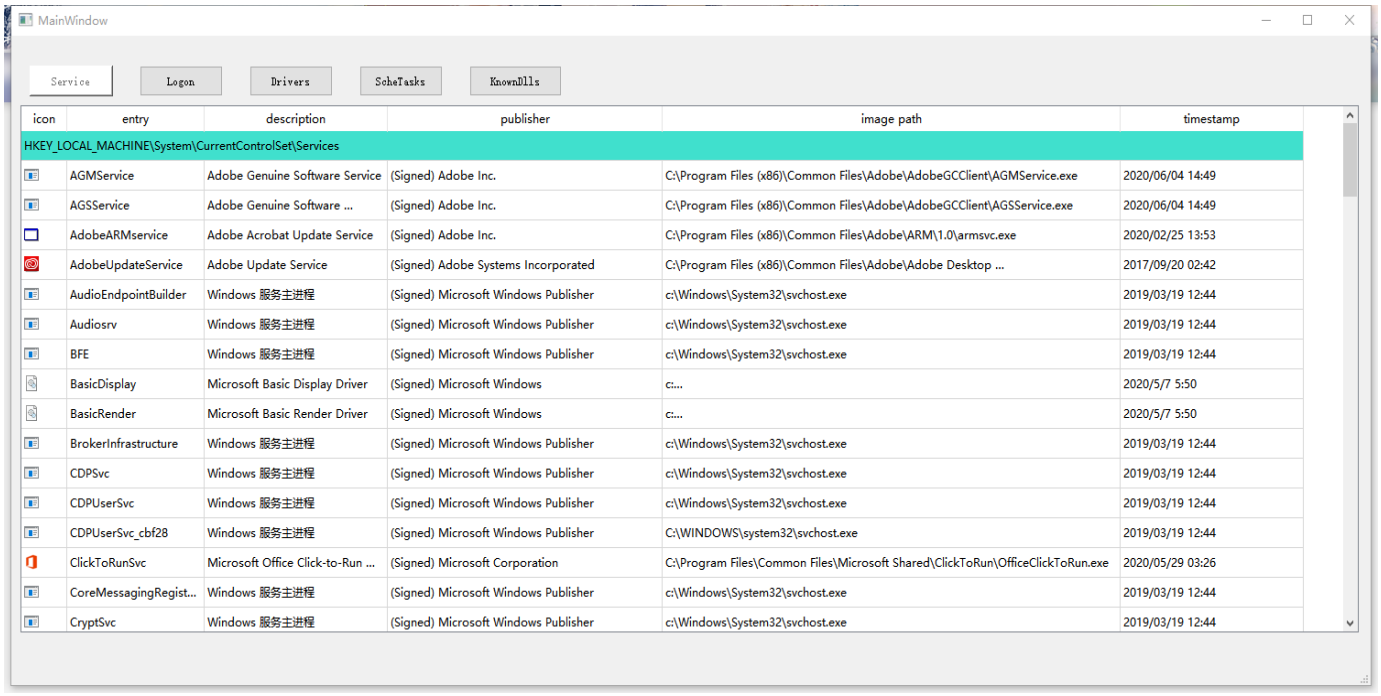


5.2 可移植性验证

- 将程序发送给其他同学，并在他们的电脑上运行。
- 运行结果分别如下：







6. 总结

6.1 主要问题与解决

- 字符串处理问题：
  - 从注册表读取的大多数 image path 并不是完整的，或者说不是规范的绝对路径，往往有环境变量、引号、反斜杠等加载在里面。

    - 我首先统计了各种不规范问题的类型，并分为两个大类：需要替换字符串和需要删除字符串。根据这两个大类分别进行处理。虽然这样做较花时间，但效果明显。
- 计划任务读取问题：
    - 在写计划任务这部分的代码时，发现编译总是有问题，COM 组件无法加载。
    - 虽然考虑过可能是编译器的问题，但担心因为这个花太多时间，所以选择了一个折中的方案：通过 Visual Studio 编写打印计划任务信息的控制台程序，在 QT Creator 中实例化 QProcess 读取控制台的输出。结果显示，这样的方案是可行的。最后和同学讨论的过程中得知，确实是编译器的问题，比较遗憾当时没能够坚定自己的判断。
- 程序异常结束问题：
    - 代码往往在运行时中断，并在 QT Creator 的应用程序输出窗口显示“程序异常结束”。
    - 这是我编程过程中经常遇到的问题。由于对 QT Creator 并不熟悉，因此只能通过注释代码、打印数据的方式排查可能出现问题的地方。经过几次这样的 debug 经历，我发现这样的问题往往是数组或列表的下标越界造成的。因此在写程序时需要重点注意数组与列表的长度，这也是我 C++ 编程不够熟练带来的问题。
- 读取文件描述、签名验证不全问题：
    - 尽管在参考网上资料后，完成了签名验证与文件描述信息读取的功能，但实际运行结果显示，有部分文件无法读取的。
    - 这一问题困扰我相当长时间，最终也未能解决，只能选择一个妥协的方案：如果自己的程序无法获得某一文件的签名验证结果，则调用 sigcheck64.exe，并读取该可执行文件的输出结果。这一方案不可避免地使程序运行时间变长，因为它不像读取计划任务信息那样只调用一次可执行文件 `schetasks.exe` 。但最终结果显示，两者结合使用后，注册表下的所有内容都能够被读取签名信息。
- 字符串类型转换问题：
    - 字符串的类型相当多，例如TChar、LPCTSTR、LPCWSTR等。而且最终函数的字符串输出往往都要转为 QString 类型。
    - 事实上这些字符串的特征跟字符串类型名称有紧密关系，例如：L 表示 long 指针，P 说明该类变量是一个指针，而 C 表示是一个常量。经过较长时间的资料查找和多次尝试不同方法后发现，主要问题在于，需要将单字节字符串转为双字节字符串，或者反向转换。因此需要先确认转换后的字符串需要的空间大小，之后才利用 `MultiByteToWideChar()` 将单字节字符串转为双字节字符串。

6.2 体会与反思

此次课程设计是我用 C++ 开发的最复杂的应用软件。自大一以后，我基本上都在使用 Python 编程。因此，一开始编程时出现了不少问题。

但总的来说，此次课程设计过程中我的思路是明确的。由于需要编写一个具有用户界面的软件，因此需要考虑有哪些合适的 IDLE；之后要分多种情况列出自启动检测结果，因此根据 button 触发显示不同的表格；查找、学习有哪些可以读取注册表的 api 接口；出现组件加载问题，但以往没有更换编译器的经验，选择读取控制台输出的方案等等。在应用开发过程中，基本上是“遇山开山，遇水搭桥”，我也在解决问题的过程中学到了不少知识。

此次课程设计，我复习了很多 C++ 编程的知识，包括指针、数组、内存泄露、字符串类型转换、面向对象编程等，并第一次使用 C++ 做出一个完整的用户图形界面，对 Windows API 以及 QT 的不少 API 都

有了一定的了解。同时，在阅读官方文档和查阅资料的过程中，我对自启动技术有了更详细的了解，提高了自己抽取关键词、搜索信息和自学的能力。

最后，非常非常感谢老师在我遇到问题时的耐心答复和指导，我也从此次课程设计中收获非常多，谢谢老师！

7.参考资料

- 设计中，为了学习 QT 的使用，我阅读了《Qt Creator快速入门 第三版》，以及 QT 的官方文档。
- 课程设计过程中我还查找了不少网上的资料，主要收藏的网址如下：
  - https://stackoverflow.com/questions/3555749/microsoft-known-dll
  - https://www.delphitips.net/2007/06/16/create-autorun-registry-key-run-application-with-windows-start/
  - https://docs.microsoft.com/en-us/windows/win32/taskschd/about-the-task-scheduler
  - https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptqueryobject
  - https://docs.microsoft.com/en-us/windows/win32/api/wintrust/nf-wintrust-winverifytrust
  - https://docs.microsoft.com/en-us/windows/win32/api/taskschd/nn-taskschd-itaskfolder
  - https://www.cnblogs.com/comor/p/10607383.html
  - https://docs.microsoft.com/zh-cn/windows/win32/taskschd/displaying-task-names-and-state--c---
  - https://support.microsoft.com/en-us/help/323809/how-to-get-information-from-authenticode-signed-executables