Статья Windows Data Structures and Callbacks, Part 1

xss.is/threads/46489

У modexp довольно миленькая картиночка в качестве хидера, почему бы её здесь не использовать?

Windows Data Structures and Callbacks, Part 1

Содержание:

Автор: modexp
Переводчик: Vism

1. Вступление

Процесс может содержать тысячи указателей на исполняемый код, некоторые из них хранятся в "неясных", но "записываемых" структурах, известных лишь только Microsoft, некоторым сторонним компаниям и плохишам, которые хотят спрятать свой вредоносный код от сканеров памяти. Этот пост существует для документирования содержания некоторых структур, а не **PoC** для демонстрации методов перенаправления и сокрытия кода, о которых, скорее всего, оригинальный автор больше не будет рассказывать. Некоторые названия полей в представленных структурах не совсем точные, но вы можете написать оригинальному автору, если считаете, что нужно что-то исправить. Структуры могут быть найдены на **MSDN** или получены **путем реверса**.

2. Dynamic Function Table List

ntdll!RtlpDynamicFunctionTable содержит `DYNAMIC_FUNCTION_TABLE` записи и callback-функции для сегмента памяти, который может быть установлен с помощью **ntdll!RtlInstallFunctionTableCallback**. **ntdll!RtlGetFunctionTableListHead** возвращает указатель на список (`PLIST_ENTRY`), а поскольку `NTDLL.dll` использует один и тот же **BA** (Base Address) для каждого процесса, то вы с легкостью сможете читать записи из удаленного процесса.

C:

```
typedef enum _FUNCTION_TABLE_TYPE {
    RF_SORTED,
    RF_UNSORTED,
    RF_CALLBACK
} FUNCTION_TABLE_TYPE;

typedef PRUNTIME_FUNCTION (CALLBACK *PGET_RUNTIME_FUNCTION_CALLBACK)
  (ULONG_PTR ControlPc, PVOID Context);

typedef struct _DYNAMIC_FUNCTION_TABLE {
    LIST_ENTRY                      Links;
    DWORD64                         TableIdentifier;
    LARGE_INTEGER                   TimeStamp;
    DWORD64                         MinimumAddress;
    DWORD64                         MaximumAddress;
    PVOID                           BaseAddress;
    PGET_RUNTIME_FUNCTION_CALLBACK  Callback;
    PCWSTR                          OutOfProcessCallbackDll;
    FUNCTION_TABLE_TYPE             Type;
    ULONG                           EntryCount;
    ULONG64                         UnknownStruct[3];  // referenced by RtlAvlInsertNodeEx
} DYNAMIC_FUNCTION_TABLE, *PDYNAMIC_FUNCTION_TABLE;
```

3. Event Tracing

Microsoft рекомендует не использовать это, но **sechost!SetTraceCallback** все еще может получать **ETW-события**. Записи по типу `EVENT_CALLBACK_ENTRY` находятся в **sechost!EtwpEventCallbackList.**

C:

```
typedef VOID (CALLBACK PEVENT_CALLBACK)(PEVENT_TRACE pEvent);

ULONG WMIAPI SetTraceCallback(
    LPCGUID         pGuid,
    PEVENT_CALLBACK EventCallback);

typedef struct _EVENT_CALLBACK_ENTRY {
    LIST_ENTRY      ListHead;
    GUID            ProviderId;
    PEVENT_CALLBACK Callback;
} EVENT_CALLBACK_ENTRY, *PEVENT_CALLBACK_ENTRY;
```

4. DLL Notifications

Можно получить уведомление о том, что DLL загружена или выгружена с помощью **ntdll!LdrRegisterDllNotification**. Используется для подключения API для **CLR** в **ClrGuard**. Записи по типу `LDR_DLL_NOTIFICATION_ENTRY` могут быть найдены в **ntdll!LdrpDllNotificationList**.

C:

```
typedef struct _LDR_DLL_LOADED_NOTIFICATION_DATA {
    ULONG           Flags;             // Reserved.
    PUNICODE_STRING FullDllName;       // The full path name of the DLL module.
    PUNICODE_STRING BaseDllName;       // The base file name of the DLL module.
    PVOID           DllBase;           // A pointer to the base address for the DLL in
memory.
    ULONG           SizeOfImage;       // The size of the DLL image, in bytes.
} LDR_DLL_LOADED_NOTIFICATION_DATA, *PLDR_DLL_LOADED_NOTIFICATION_DATA;

typedef struct _LDR_DLL_UNLOADED_NOTIFICATION_DATA {
    ULONG           Flags;             // Reserved.
    PUNICODE_STRING FullDllName;       // The full path name of the DLL module.
    PUNICODE_STRING BaseDllName;       // The base file name of the DLL module.
    PVOID           DllBase;           // A pointer to the base address for the DLL in
memory.
    ULONG           SizeOfImage;       // The size of the DLL image, in bytes.
} LDR_DLL_UNLOADED_NOTIFICATION_DATA, *PLDR_DLL_UNLOADED_NOTIFICATION_DATA;

typedef VOID (CALLBACK *PLDR_DLL_NOTIFICATION_FUNCTION)(
    ULONG                       NotificationReason,
    PLDR_DLL_NOTIFICATION_DATA  NotificationData,
    PVOID                       Context);

typedef union _LDR_DLL_NOTIFICATION_DATA {
    LDR_DLL_LOADED_NOTIFICATION_DATA   Loaded;
    LDR_DLL_UNLOADED_NOTIFICATION_DATA Unloaded;
} LDR_DLL_NOTIFICATION_DATA, *PLDR_DLL_NOTIFICATION_DATA;

typedef struct _LDR_DLL_NOTIFICATION_ENTRY {
    LIST_ENTRY                      List;
    PLDR_DLL_NOTIFICATION_FUNCTION Callback;
    PVOID                           Context;
} LDR_DLL_NOTIFICATION_ENTRY, *PLDR_DLL_NOTIFICATION_ENTRY;

typedef NTSTATUS(NTAPI *_LdrRegisterDllNotification) (
    ULONG                           Flags,
    PLDR_DLL_NOTIFICATION_FUNCTION NotificationFunction,
    PVOID                           Context,
    PVOID                           *Cookie);

typedef NTSTATUS(NTAPI *_LdrUnregisterDllNotification)(PVOID Cookie);
```

5. Secure Memory

Ring-0 драйверы могут "обезопасить" пользовательскую память используя **ntoskrnl!MmSecureVirtualMemory**. Это может предотвратить освобождение памяти или "усиление" защиты страницы, т.е `PAGE_NOACCESS`. Для мониторинга изменений разработчики могут использовать **AddSecureMemoryCacheCallback**. Записи по типу `RTL_SEC_MEM_ENTRY` находятся в **ntdll!RtlpSecMemListHead**.

C:

```
typedef BOOLEAN (CALLBACK *PSECURE_MEMORY_CACHE_CALLBACK)(PVOID, SIZE_T);

typedef struct _RTL_SEC_MEM_ENTRY {
    LIST_ENTRY                    Links;
    ULONG                         Revision;
    ULONG                         Reserved;
    PSECURE_MEMORY_CACHE_CALLBACK Callback;
} RTL_SEC_MEM_ENTRY, *PRTL_SEC_MEM_ENTRY;
```

6. Configuration Manager (CM)

Процесс может регистрировать события типа Plug & Play при помощи **cfgmgr32!CM_Register_Notification**. Microsoft рекомендует в старых системах (до Windows 7) использовать **RegisterDeviceNotification**, но автор статьи не рассматривал эту функцию. Записи уведомлений по типу `_HCMNOTIFICATION` находятся в **cfgmgr32!EventSystemClientList**. `_CM_CALLBACK_INFO` - структура, которая может быть отправлена в `\Device\DeviceApi\CMNotify`, когда процесс регистрирует callback. Как видно из поля `WnfSubscription`, он использует **Windows Notification Facility** (WNF), для получения ивентов.

C:

```
typedef DWORD (CALLBACK *PCM_NOTIFY_CALLBACK)(
    _In_     HCMNOTIFICATION       hNotify,
    _In_opt_ PVOID                 Context,
    _In_     CM_NOTIFY_ACTION      Action,
    _In_     PCM_NOTIFY_EVENT_DATA EventData,
    _In_     DWORD                 EventDataSize);

typedef struct _CM_CALLBACK_INFO {
    WCHAR              ModulePath[MAX_PATH];
    CM_NOTIFY_FILTER   EventFilter;
};

typedef struct _tagHCMNOTIFICATION {
    USHORT                  Signature;             // 0xF097
    SRWLOCK                 SharedLock;
    CONDITION_VARIABLE      ConditionVar;
    LIST_ENTRY              EventSystemClientList;
    LIST_ENTRY              EventSystemPendingClients;
    BOOL                    Active;
    BOOL                    InProgress;
    CM_NOTIFY_FILTER        EventFilter;
    PCM_NOTIFY_CALLBACK     Callback;
    PVOID                   Context;
    HANDLE                  NotifyFile;    // handle for \Device\DeviceApi\CMNotify
    PWNF_USER_SUBSCRIPTION  WnfSubscription;
    LIST_ENTRY              Links;
} _HCMNOTIFICATION, *_PHCMNOTIFICATION;
```

7. Vectored Exception Handling (VEH)

Когда выполняется **kernelbase!KernelBaseBaseDllInitialize**, он устанавливет обработчик исключений (англ. exception handler) **kernelbase!UnhandledExceptionFilter** через **SetUnhandledExceptionFilter**. Однако, если после этого не был установлен **VEH**, то этот обработчик будет выполняется на "верхнем уровне" для любых возникающих неисправностей. **VEH** callback'и, установленные при помощи **AddVectoredExceptionHandler** или **AddVectoredContinueHandler** находятся в **ntdll!LdrpVectorHandlerList**.

C:

```
// vectored handler list
typedef struct _RTL_VECTORED_HANDLER_LIST {
    SRWLOCK                    Lock;
    LIST_ENTRY                  List;
} RTL_VECTORED_HANDLER_LIST, *PRTL_VECTORED_HANDLER_LIST;

// exception handler entry
typedef struct _RTL_VECTORED_EXCEPTION_ENTRY {
    LIST_ENTRY                  List;
    PULONG_PTR                  Flag;           // some flag related to CFG
    ULONG                       RefCount;
    PVECTORED_EXCEPTION_HANDLER VectoredHandler;
} RTL_VECTORED_EXCEPTION_ENTRY, *PRTL_VECTORED_EXCEPTION_ENTRY;
```

8. Windows Error Reporting (WER)

Windows предоставляет API для восстановления приложений, дампа памяти приложения и генерации отчетов через **WER-службу**. Настройки **WER** для процесса могут находиться в **Process Environment Block** (PEB) в **WerRegistrationData**.

8.1 PEB Header Block

Автор статьи обсудит структуры отдельно, но только те, которые не являются таковыми. `Signature` (поле структуры) устанавливается внутри **kernelbase!WerpInitPEBStore** и содержит строку "PEB_SIGNATURE". **AppDataRelativePath** задается при помощи **WerRegisterAppLocalDump**. **kernelbase!RegisterApplicationRestart** может быть использован для установки **RestartCommandLine**, который используется в качестве командной строки, когда процесс должен быть... перезапущен.

C:

```
typedef struct _WER_PEB_HEADER_BLOCK {
    LONG                 Length;
    WCHAR                Signature[16];
    WCHAR                AppDataRelativePath[64];
    WCHAR                RestartCommandLine[RESTART_MAX_CMD_LINE];
    WER_RECOVERY_INFO    RecoveryInfo;
    PWER_GATHER          Gather;
    PWER_METADATA        MetaData;
    PWER_RUNTIME_DLL     RuntimeDll;
    PWER_DUMP_COLLECTION DumpCollection;
    LONG                 GatherCount;
    LONG                 MetaDataCount;
    LONG                 DumpCount;
    LONG                 Flags;
    WER_HEAP_MAIN_HEADER MainHeader;
    PVOID                Reserved;
} WER_PEB_HEADER_BLOCK, *PWER_PEB_HEADER_BLOCK;
```

8.2 Recovery Information

Callback восстановления может быть установлен с помощью **kernel32!RegisterApplicationRecoveryCallback**. **kernelbase!GetApplicationRecoveryCallback** прочтет поля `Callback`, `Parameter`, `PingInterval` и `Flags` из удаленного процесса. **kernel32!ApplicationRecoveryFinished** может быть "читать", если ивент `Finished` просигнализировал о том, что он завершен. **ApplicationRecoveryInProgress** постарается определить, сигнализирует ли ивент `InProgress`. `Started` - хэндл, но автор не уверен, для чего он.

C:

```
typedef struct _WER_RECOVERY_INFO {
    ULONG                Length;
    PVOID                Callback;
    PVOID                Parameter;
    HANDLE               Started;
    HANDLE               Finished;
    HANDLE               InProgress;
    LONG                 LastError;
    BOOL                 Successful;
    DWORD                PingInterval;
    DWORD                Flags;
} WER_RECOVERY_INFO, *PWER_RECOVERY_INFO;
```

8.3 Gathers

Как некоторая часть отчета, созданного **WER**, **kernelbase!WerRegisterMemoryBlock** предоставляет информацию о сегменте памяти, который должна быть включен в отчет. Можно также исключить сегмент памяти используя **kernelbase!WerRegisterExcludedMemoryBlock**, который устанавливает 15 бит в поле `Flags` в структуре `WER_GATHER`. Файлы, которые могли бы быть исключены из отчета, можно сохранить через **kernelbase!WerRegisterFile**.

C:

```
typedef struct _WER_FILE {
    USHORT               Flags;
    WCHAR                Path[MAX_PATH];
} WER_FILE, *PWER_FILE;

typedef struct _WER_MEMORY {
    PVOID                Address;
    ULONG                Size;
} WER_MEMORY, *PWER_MEMORY;

typedef struct _WER_GATHER {
    PVOID                Next;
    USHORT               Flags;
    union {
      WER_FILE           File;
      WER_MEMORY         Memory;
    } v;
} WER_GATHER, *PWER_GATHER;
```

8.4 Custom Meta Data

Приложения могут регистрировать пользовательские мета-данные используя **kernelbase!WerRegisterCustomMetadata**.

C:

```
typedef struct _WER_METADATA {
    PVOID                Next;
    WCHAR                Key[64];
    WCHAR                Value[128];
} WER_METADATA, *PWER_METADATA;
```

8.5 Runtime DLL

Разработчики могут настраивать процесс создания отчетов, для чего предназначен **kernelbase!WerRegisterRuntimeExceptionModule**. Он вставляет путь до DLL в данные (регистрационные), которые загружаются через `werfault.exe` при

возникновении исключений. В структуре `WER_RUNTIME_DLL`, `MAX_PATH` используется для `CallbackDllPath`, но корректная длина для структуры и DLL должна быть прочитана из поля `Length`.

C:

```
typedef struct _WER_RUNTIME_DLL {
    PVOID               Next;
    ULONG               Length;
    PVOID               Context;
    WCHAR               CallbackDllPath[MAX_PATH];
} WER_RUNTIME_DLL, *PWER_RUNTIME_DLL;
```

8.6 Dump Collections

Если требуется более одного процесса для дампа, то приложение может использовать **kernelbase!WerRegisterAdditionalProcess** для указания идентификаторов процессов и потоков. Автор не против, если вы его поправите, но похоже, что API разрешает только один поток на процесс.

C:

```
typedef struct _WER_DUMP_COLLECTION {
    PVOID               Next;
    DWORD               ProcessId;
    DWORD               ThreadId;
} WER_DUMP_COLLECTION, *PWER_DUMP_COLLECTION;
```

8.7 Heap Main Header

Наконец, главный заголовок кучи, используемый для динамической аллокации памяти для **WER-структур**. Поле `Signature` должно содержать строку "HEAP_SIGNATURE". `Mutex` здесь просто для "уникального" доступа во время аллокации. Определение `FreeHeap` в этой структуре может быть не точным, но, похоже, оно используется для улучшения производительности аллокации. Вместо того, чтоб запрашивать каждый раз у ОС новый блок памяти, **WER-функции** могут по возможности использовать этот блок.

C:

```
typedef struct _WER_HEAP_MAIN_HEADER {
    WCHAR                Signature[16];
    LIST_ENTRY           Links;
    HANDLE               Mutex;
    PVOID                FreeHeap;
    ULONG                FreeCount;
} WER_HEAP_MAIN_HEADER, *PWER_HEAP_MAIN_HEADER;
```

Служба **WER** может вполне оправданно стать точкой для **повышения привелигий**. Это можно использовать для кражи конфиденциальной информации путем изменения информации в **настройках реестра**. Атакующий может быть способен **сдампить процесс** и **отправить отчет на подконтрольный ему сервер** с помощью настройки **CorporateWERServer.** Атакующий также может использовать собственный открытый ключ шифрования этих данных и предотвращения восстановления того, что именно собирается с жертвы. Автор подмечает, что это все конечно чисто гипотетически и не знает можно ли это использовать в таких целях.