SHIM me what you got: Manipulating Shim and Office for Code Injection

Ron Ben Yizhak & David Shandalov

About Us

Ron Ben Yizhak

Security Researcher @ Deep Instinct

Interested in malware campaigns, attack vector and evasion techniques

Enjoys rock climbing and volleyball



David Shandalov

Security Researcher @ Deep Instinct

Likes reverse-engineering, malware analysis and exploitation

Enjoys flying and aviation



Agenda

1 Intro

Overview the shim framework and known ways to abuse it

2 RPC in Office

Attack surface research on RPC servers in MS Office

3 App Compatibility

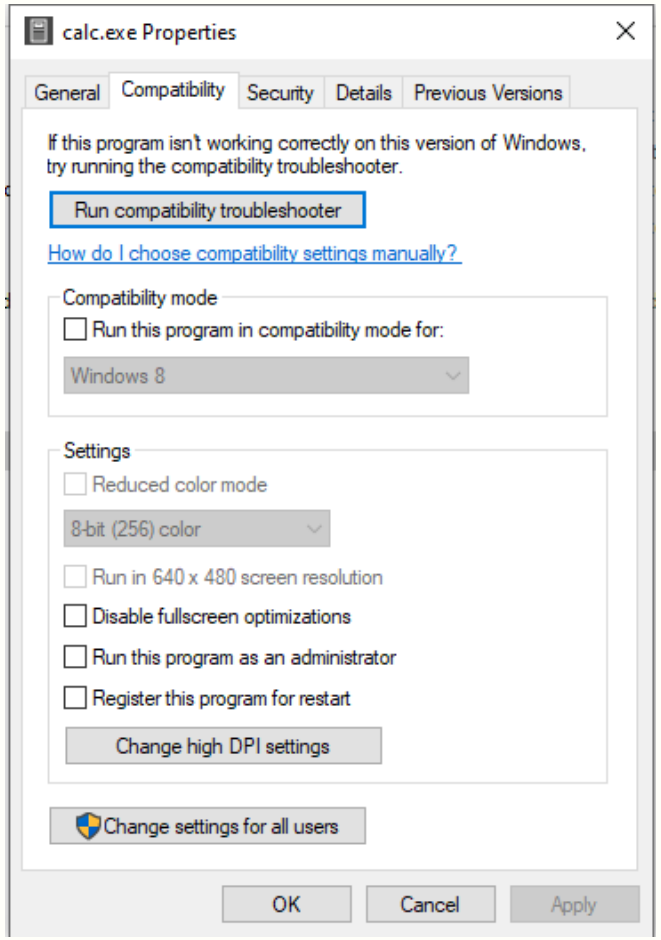
Reverse engineer the mechanism to renovate the attack

4 Conclusion

Detection methods and further research

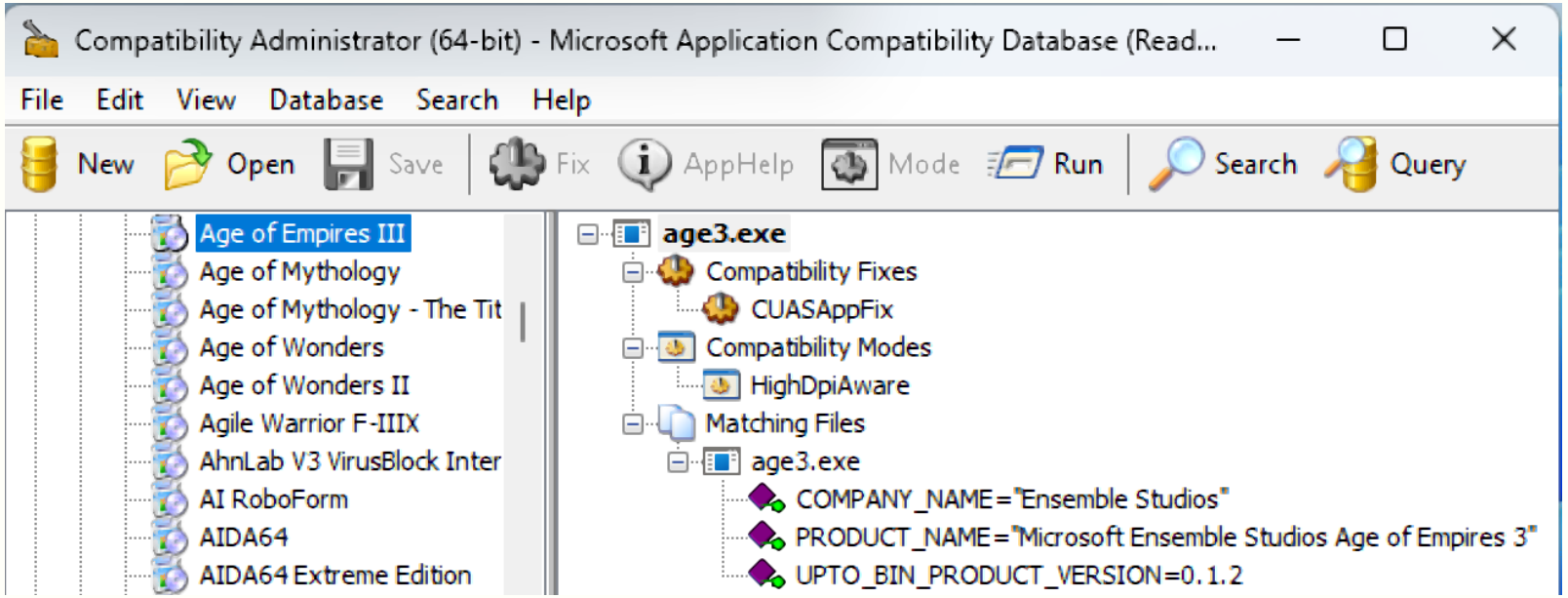
Intro - App Compatibility

- Framework provided by Microsoft to ensure old applications can run smoothly on new OS versions
- It includes various compatibility modes, shims, and other mechanisms that help legacy applications function correctly
- It addresses issues related to OS changes:
  - APIs
  - security features
  - system configurations



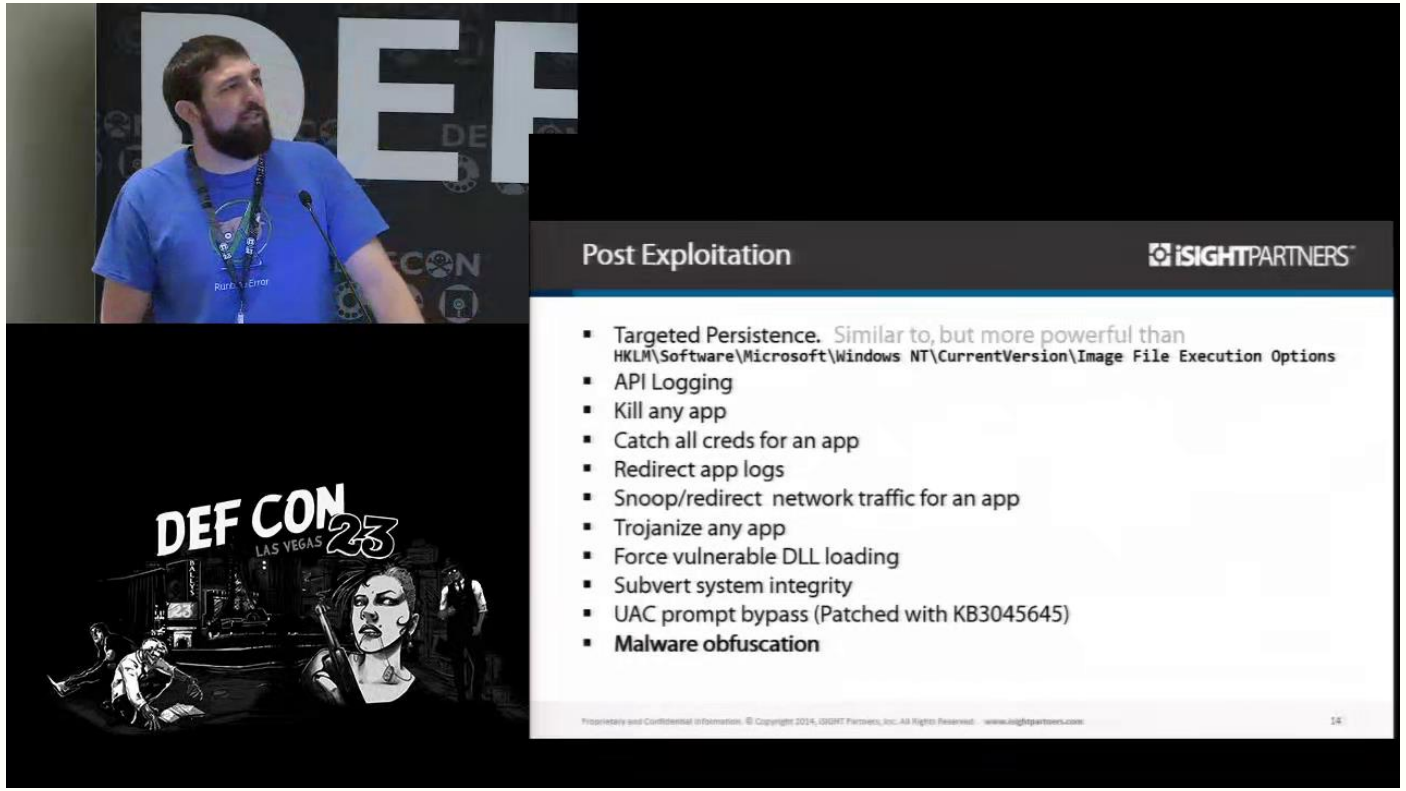
Intro - App Compatibility

- This framework uses SDB files, or "Shim Database" files, to store compatibility fixes
- The main one is C:\Windows\apppatch\sysmain.sdb
- These files are managed by the Compatibility Administrator tool
- Custom SDB files are registered under these paths:
  - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom
  - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\InstalledSDB



Intro - App Compatibility

- SDB files are a known attack vectors, and can be abused for code injection and API hooking
- EDR solutions can detect malicious SDB files by monitoring the related registry keys



DEF CON 23 - Sean Pierce - Abusing native Shims for Post Exploitation



Intro - Office

- Office is a great candidate for an attack surface research
- It is very popular
- It is a complex software that is made of several components
- It has backward compatibility for many features



Agenda

1 Intro

Overview the shim framework and known ways to abuse it

2 RPC in Office

Attack surface research on RPC servers in MS Office

3 App Compatibility

Reverse engineer the mechanism to renovate the attack

4 Conclusion

Detection methods and further research

RPC Interfaces in Office

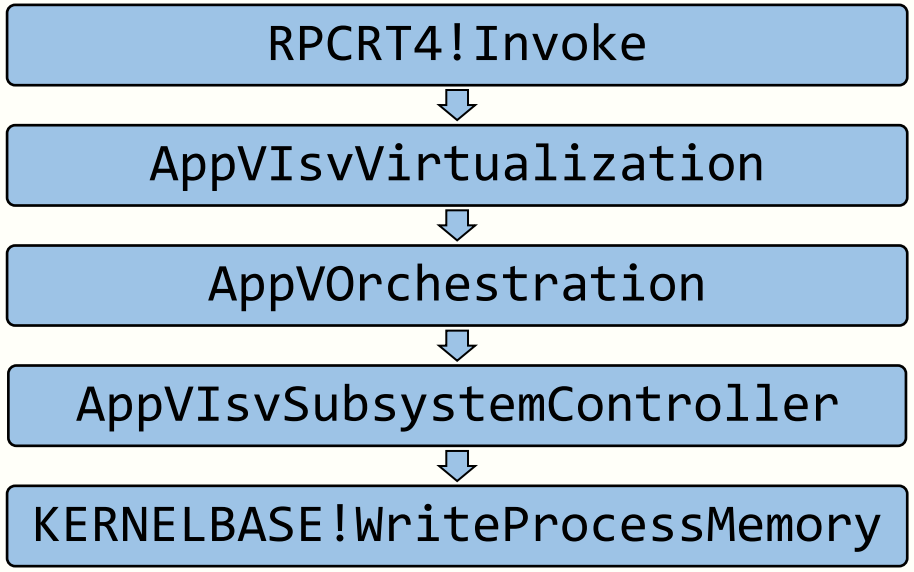
- Looking for running RPC servers led to the service ClickToRunSvc
- The exposed interfaces are implemented in the following files:
  - OfficeClickToRun.exe
  - AppVIsvVirtualization.dll
  - AppVIsvSubsystemController.dll

Interface Id	Procedure Count	File Path
469d3a0e-e164-422e-a662-9cbe0621407e	60	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeClickToRun.exe
afa8bd80-7d8a-11c9-bef4-08002b102989	5	C:\Windows\System32\RPCRT4.dll
18f70770-8e64-11cf-9af1-0020af6e72f4	5	C:\Windows\System32\combase.dll
69c09ea0-4a09-101b-ae4b-08002b349a02	0	C:\Windows\System32\combase.dll
048cf666-ab42-42b4-8975-1357018decb3	6	C:\Windows\System32\WS2_32.dll
4f32adc8-6052-4a04-8701-293ccf2096f0	1	C:\Windows\SYSTEM32\SspiCli.dll
cad784cb-4c1b-4d96-b8f7-4716b568b13c	83	C:\Windows\SYSTEM32\WININET.dll
3473dd4d-2e88-4006-9cba-22570909dd10	9	C:\Windows\SYSTEM32\WINHTTP.dll
824d8d77-a27f-4915-a536-36e9283dce29	11	C:\Windows\SYSTEM32\WINHTTP.dll
09c76598-1491-4810-bbb0-7f403a2ab7ea	3	C:\Windows\SYSTEM32\WINHTTP.dll
d99d2add-4f00-41b8-bf99-bb03127f015f	2	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvVirtualization.dll
8c7fbdb0-8513-44f9-a8b1-1a3b49322bf4	1	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll
d614caa0-0d0e-4b3f-a285-ce1c560b3d7d	2	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll
4b183cf6-affd-4872-9da2-7564b683d027	6	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll
8d17061c-534a-4f1b-bd77-f615421cf379	4	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll
6d809348-7e6c-41b9-91bc-630fe5503d66	1	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll
44e10347-37a0-494c-871c-fb90f7145742	10	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll
edce686d-acae-4a2a-8945-24489443c35e	1	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\AppVIsvSubsystemController.dll

RPC Interfaces in Office

- AppVIsvVirtualization.dll contains a string indicating DLL injection
- It is printed by a function exposed through the RPC interface
- The service was monitored for calling WinAPI used for DLL injection
- Launching Winword.exe caused the service to call **`WriteProcessMemory`**

```
180066FC0 aCouldNotInject:                        ; DATA XREF: sub_180007D6C+17A↑o
180066FC0                 text "UTF-16LE", 'Could not inject subsystem dll to child process, pr'
180067026                 text "UTF-16LE", 'ocess id %3%, module %2%, error code %1%',0
180067078                 align 20h
```



AppVIsvVirtualization.dll

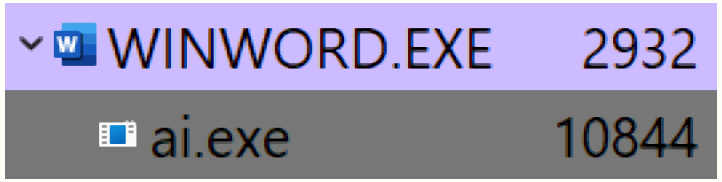
- Microsoft Application Virtualization Client Virtualization Manager Component
- It is an RPC server that exposes two undocumented methods
- The RPC client is AppvIsvSubsystems64.dll
- The public symbols reveal the names of the RPC methods
  - `virtman_notification_server_notify_new_process`
  - `virtman_notification_server_notify_new_child_process`

```
PS C:\WINDOWS\system32> ls "C:\Program Files\Common Files\microsoft
shared\ClickToRun\\*" -recurse | Get-RpcServer -ParseClients -DbgHelpPath
$Env:DbgHelp | ? {($_.InterfaceId -eq 'd99d2add-4f00-41b8-bf99-bb03127f015f')} |
Select FilePath

FilePath
--------
C:\Program Files\Common Files\microsoft shared\ClickToRun\AppvIsvSubsystems64.dll
C:\Program Files\Common Files\microsoft shared\ClickToRun\AppVIsvVirtualization.dll
```

The Injection Process - Client Side

- To understand how the injection is done, we looked at what the client sends
- Before the RPC call is made, a child process is created in a suspended state
- The parameters are the pid of ai.exe and the string *AppvIsvSubsystems64.dll*

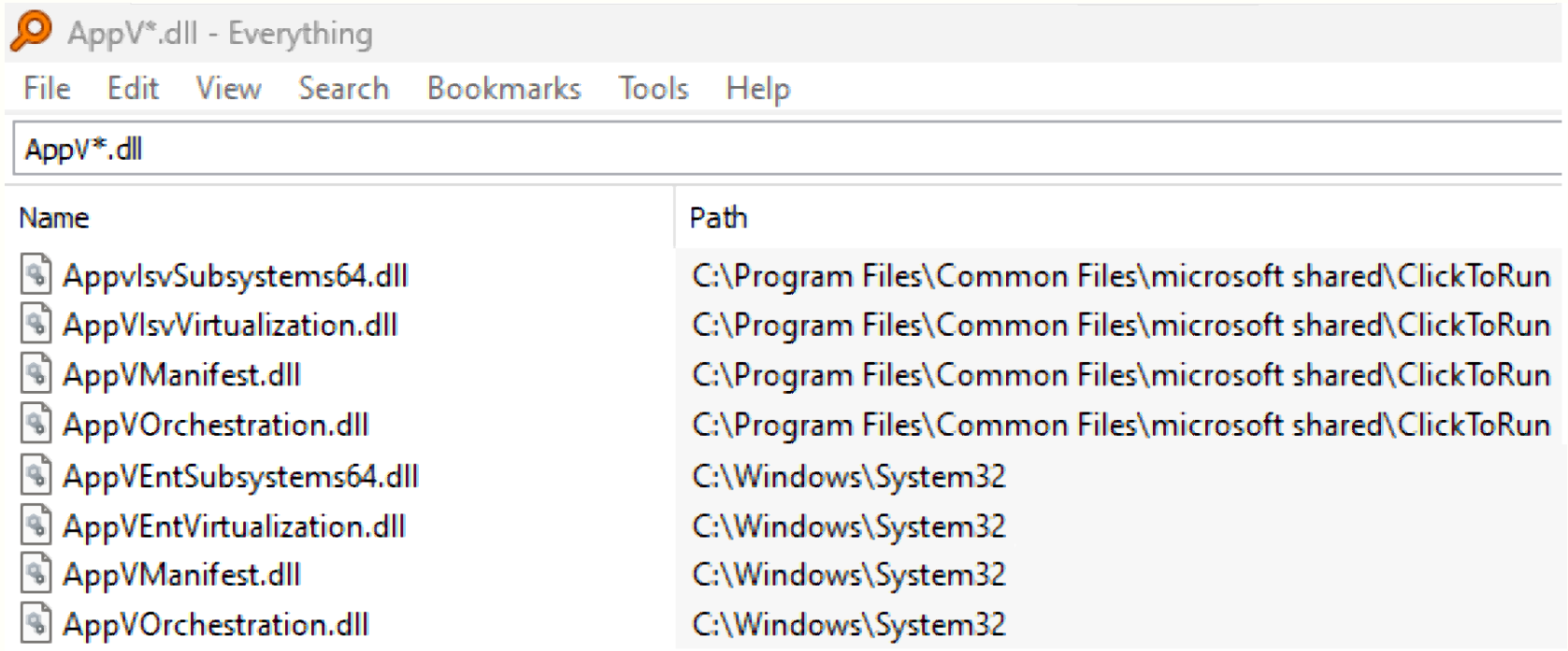


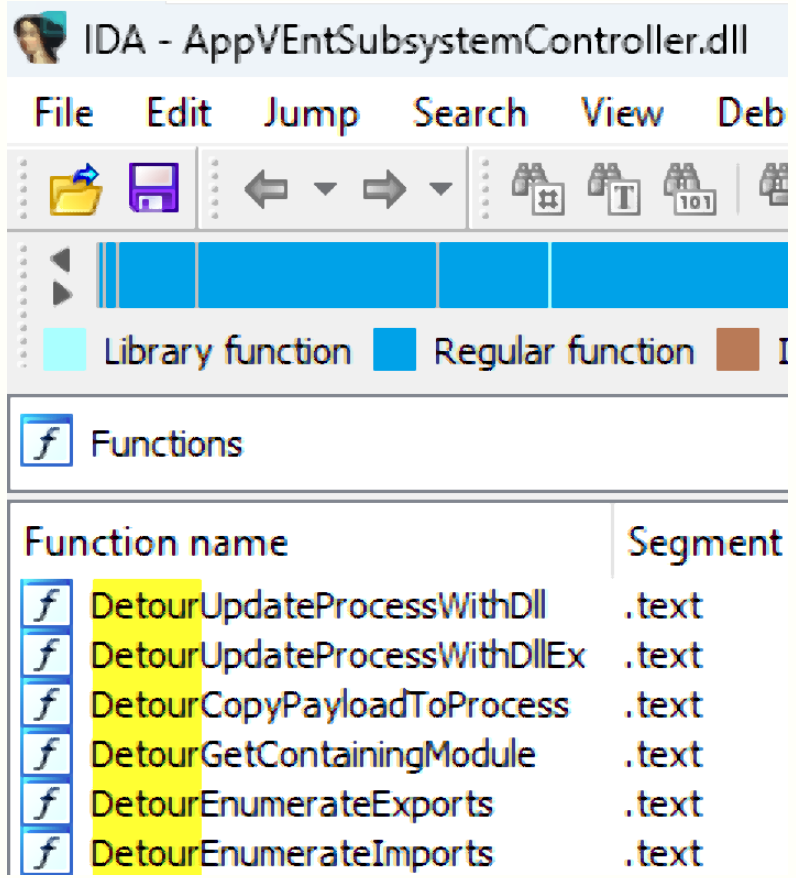

```
typedef struct _PATH_STRUCT
{
    DWORD PathLength;
    [size_is(PathLength), unique] const wchar_t* Path;
} PATH_STRUCT;

int notify_new_child_process([in] int Pid, [in] PATH_STRUCT* PathStruct)
```

The Injection Process - Server Side

- DLL files like the ones used by ClickToRunSvc were found in C:\windows\system32
- Those files have public symbols that show they use the Detours library
- The injection is done by calling `DetourUpdateProcessWithDll` & `DetourCopyPayloadToProcess`
- These functions patch the IAT of the target process





The Injection Process

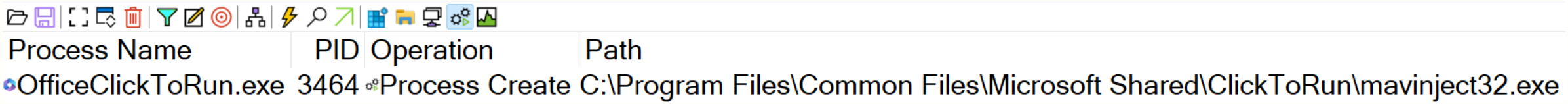
- The following changes are made to the memory of ai.exe:
  - New page is allocated, and two PE files are written into it, which are used by the Detours library
  - The optional header is modified - the checksum field is zeroed, and the virtual address and size of the import table are modified
  - Another import table is allocated in a new page, which includes an additional descriptor

```
0:001> !dh ai_exe -a
[snip]
OPTIONAL HEADER VALUES
     20B magic #
       0 checksum
[snip]
  _IMAGE_IMPORT_DESCRIPTOR 00007ff63f380000
    C:\Program Files\Microsoft Office\root\Client\AppVIsvSubsystems64.dll
      00007FF63F380100 Import Address Table
      00007FF63F3800F0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

       00007FF9735874D0      Ordinal     1
```

The Injection Process

- The suspended process is resumed after the RPC call
- The process is not fully initialized yet, so loader uses the modified import table
- AppvIsvSubsystems64.dll is injected into the process
- OfficeClickToRun.exe doesn't modify the memory of 32-bit processes directly
- It launches mavinject32.exe, which does the same injection



Process Name	PID	Operation	Path
OfficeClickToRun.exe	3464	Process Create	C:\Program Files\Common Files\Microsoft Shared\ClickToRun\mavinject32.exe

Abusing ClickToRunSvc

- Security products exclude popular software to reduce false positives
- Forcing OfficeClickToRun.exe to inject a DLL might result in a detection bypass
- We can imitate the RPC call to inject AppvIsvSubsystems64.dll
- Can we inject another DLL?
- Writing a DLL file to the Office directory is very suspicious
- It also requires administrator privileges



Abusing ClickToRunSvc

- Changing the string to “C:\Temp\Injected.dll” results in an attempt to load “C:\Program Files\Common Files\microsoft shared\ClickToRun\C:\Temp\Injected.dll”
- There are no sanity checks on the string
- It is vulnerable to directory traversal
- The file lookup can be manipulated into a completely different directory

```
0:001> !dh notepad -a
[snip]
  _IMAGE_IMPORT_DESCRIPTOR 00007ff6029b0000
    C:\Program Files\Microsoft Office\root\Client\..\..\..\..\temp\Injected.dll
      00007FF6029B0420 Import Address Table
      00007FF6029B0410 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

       00007FF9913A1030      Ordinal     1
```

Abusing ClickToRunSvc

- The modified IAT will be used only if the process isn't fully initialized
- We cannot inject into initialized processes
- We found a new technique to perform DLL injection that is done by a benign application
- What more can be achieved?
- The injection is done by a process that runs as "NT AUTHORITY\SYSTEM"
- The client can run with low privileges
- Can we abuse this service for privilege escalation?

Security mechanisms in ClickToRunSvc

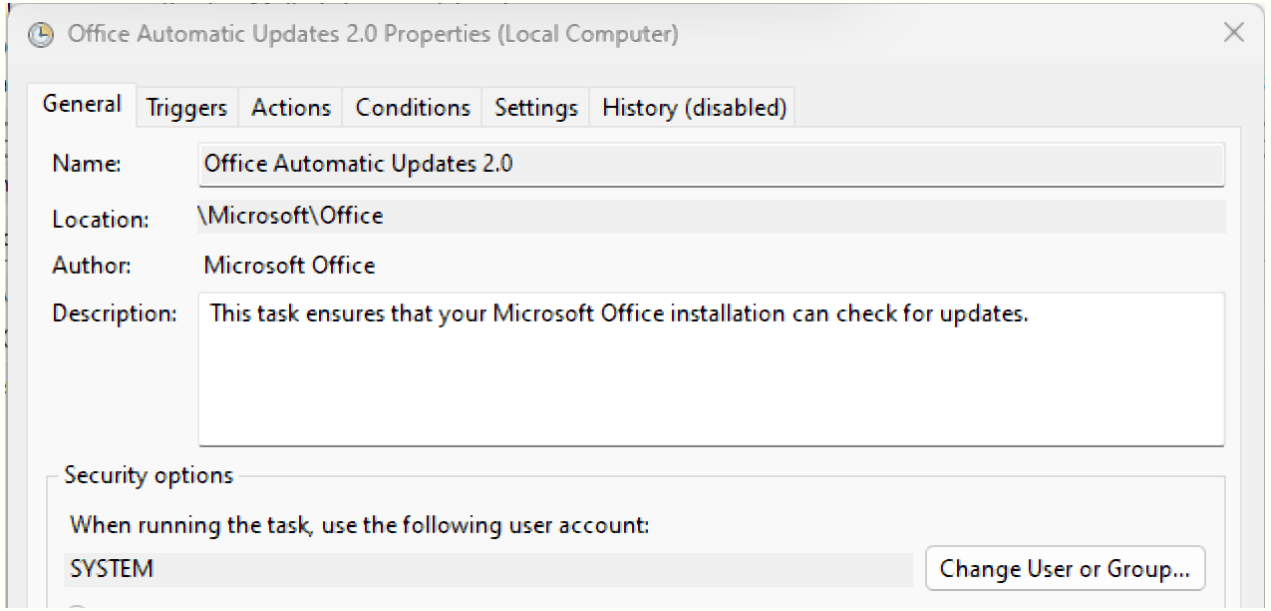
- ClickToRunSvc impersonates the RPC client before the injection
- The thread handling the RPC request will use the same token as the client
- It verifies that the client is allowed to access the remote process
- The call to **`OpenProcess`** made by AppVIsvSubsystemController will fail
- Can we escalate from admin to NT AUTHORITY\SYSTEM?

```
DWORD __fastcall sub_180009DE4(DWORD *PidPointer, __int64 a2, __int64 a3, unsigned int a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  Wow64Process = a2;
  processHandle = OpenProcess(PROCESS_QUERY_INFORMATION, 0, *PidPointer);
  if ( !processHandle )
    return GetLastError() | 0x20000;
  [snip]
}
```

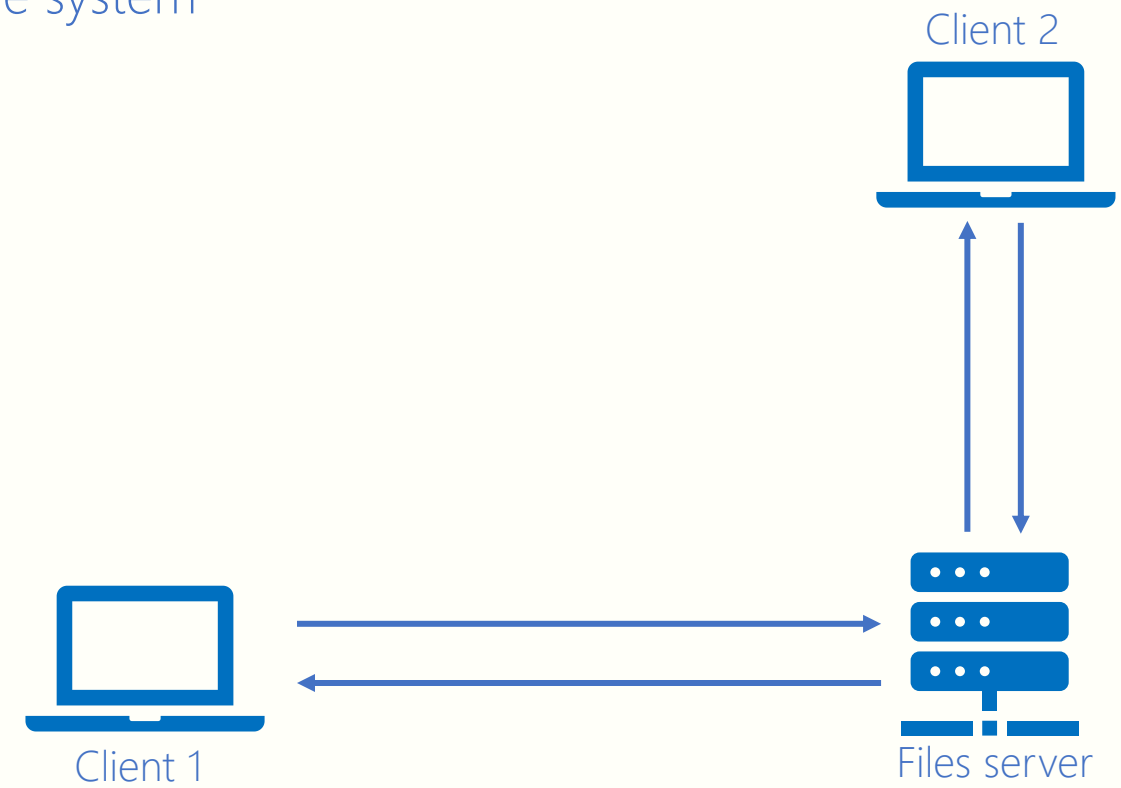
Launching suspended process as SYSTEM

- Creation of suspended processes was looked for using API monitoring
- This led to the task scheduler service
- Many scheduled tasks are launched as SYSTEM, even some belong to Office
- Forcing ClickToRunSvc to inject a DLL into another Office process isn't suspicious
- We need find a way to hold the execution of the task scheduler service
- One way we can do that is by using an opportunistic lock



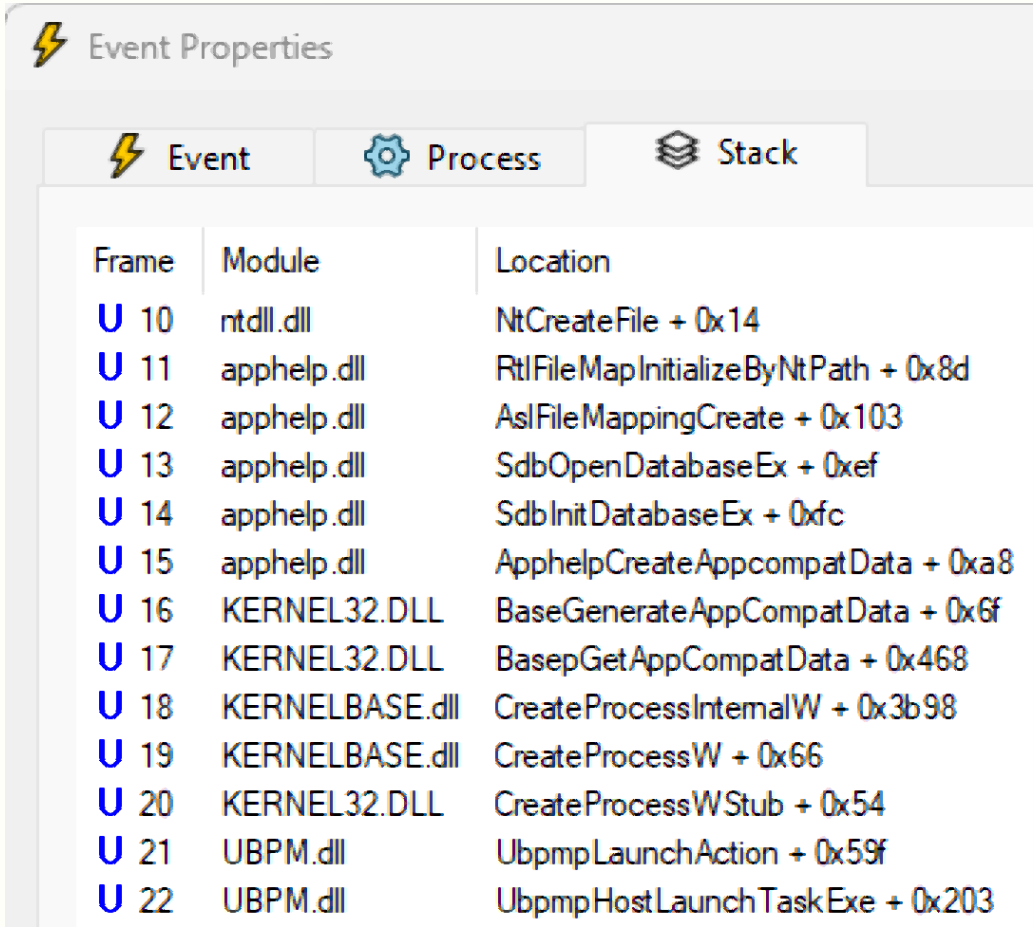
Opportunistic Locks

- Mechanism used in networked file systems to keep data coherency
- Client can obtain a lock for a file and be notified when it breaks
- It allows the client to perform operations on the file without interference from other clients
- Locks can be requested by processes from the local file system



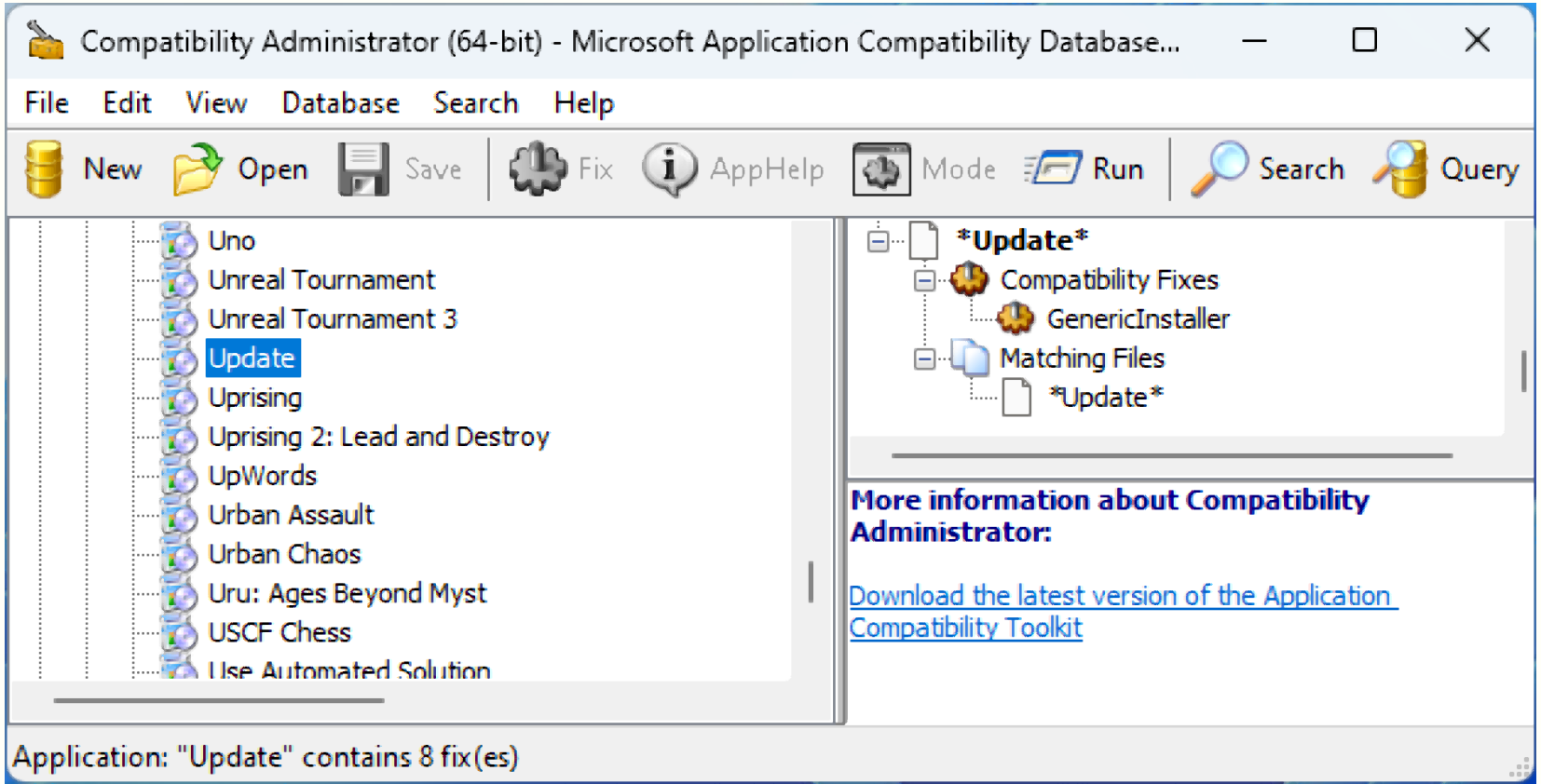
Opportunistic Locks

- We need to find a file that accessed by the task scheduler after the process is created but before it is resumed
- Requesting an oplock for this file will stop the service from resuming the process
- While we hold the lock, we can modify the suspended process
- Monitoring file access while launching scheduled tasks led to C:\Windows\apppatch\sysmain.sdb



App Compatibility

- We need to find a scheduled task that executes a file with app compatibility settings
- Every task was executed to check which one caused the service to read the SDB file
- Tasks that matched launched C:\Program Files (x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe



App Compatibility

- The SDB file was read the first time an executable ran since the machine booted, even if no fix is needed
- After that, in all subsequent runs, the SDB file wasn't read
- We want to find a way to guarantee the SDB file will be read any time an executable runs
- Registering a custom SDB file is too noisy
- Looking for files related to app compatibility led to apphelp.dll
- The main API it uses is `NtApphelpCacheControl`

NtApphelpCacheControl

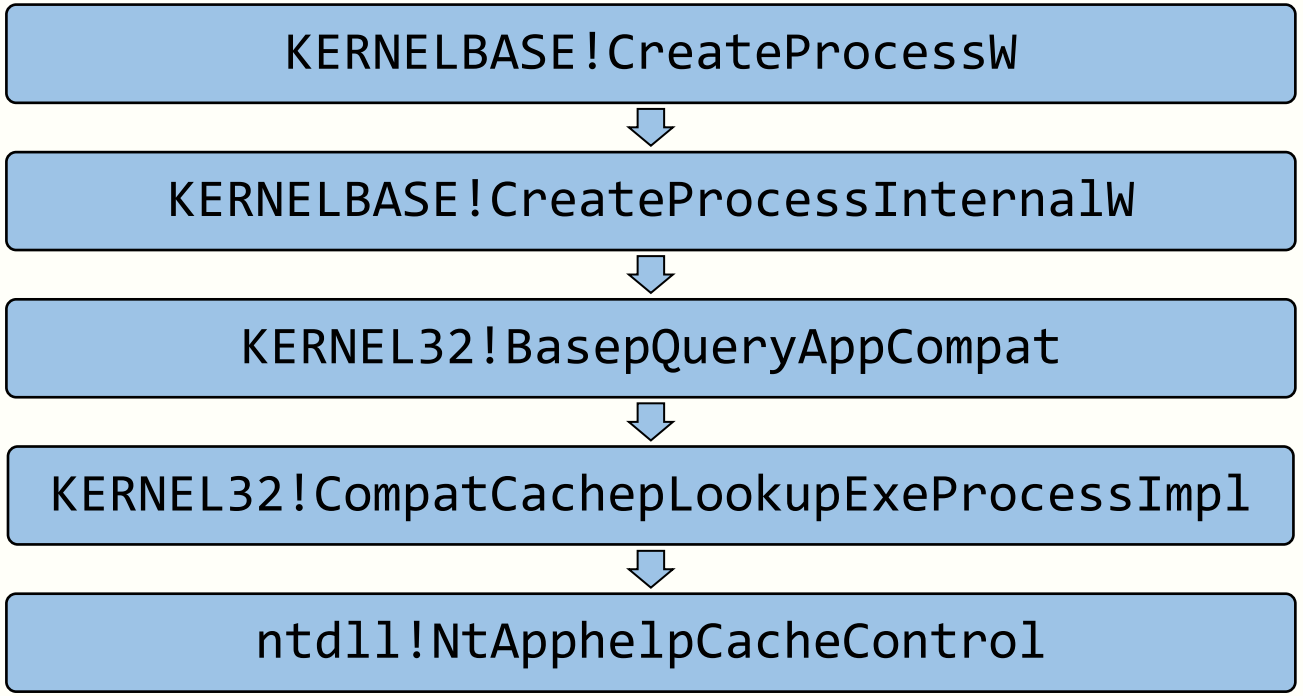
- We discovered that the symbols of combase.dll contain information about the API!

```
__int64 __fastcall CompatCachepLookupCdb(const wchar_t *Name, unsigned int a2)
{
  unsigned int v3; // ebx
  wchar_t *v4; // rax
  const wchar_t *v5; // rax
  _AHC_SERVICE_DATA ServiceData; // [rsp+20h] [rbp-198h] BYREF
  unsigned int Data; // [rsp+1C8h] [rbp+10h] BYREF

  Data = a2;
  memset_0(&ServiceData, 0, sizeof(ServiceData));
  v3 = 0;
  Data = 0;
  v4 = wcsrchr(Name, 0x5Cu);
  if ( v4 )
    v5 = v4 + 1;
  else
    v5 = Name;
  if ( RtlInitUnicodeStringEx(&ServiceData.LookupCdb.Name, v5) < 0 )
    return 0i64;
  ServiceData.ParamsOutSize = 4;
  ServiceData.ParamsOut = &Data;
  if ( g_CompatCacheServiceApi(ApphelpCacheServiceLookupCdb, &ServiceData) >= 0 && (Data & 2) != 0 )
    return 1;
  return v3;
}
```

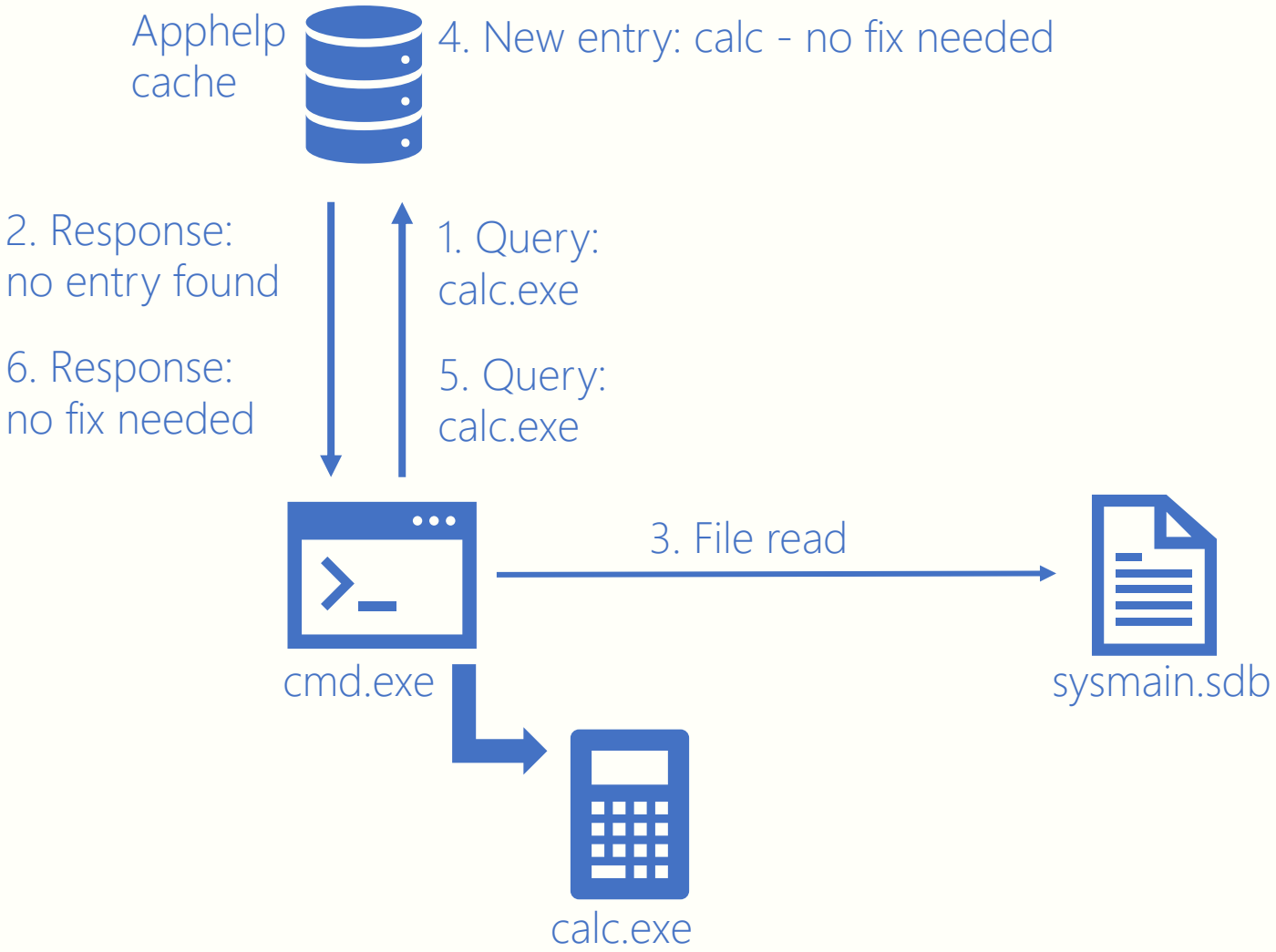
NtApphelpCacheControl

- Extracting the definitions revealed all the various operations of the API
- The calls to the API in other DLL files were made much clearer
- `CreateProcess` performs the operation `ApphelpCacheServiceLookupProcess` behind the scenes



NtApphelpCacheControl

- If we could remove this entry from the cache, the SDB file will be read again



NtApphelpCacheControl

- Both kernel32.dll and apphelp.dll perform the "remove" operation
- After this call, the execution of the file will once again trigger reading the SDB file

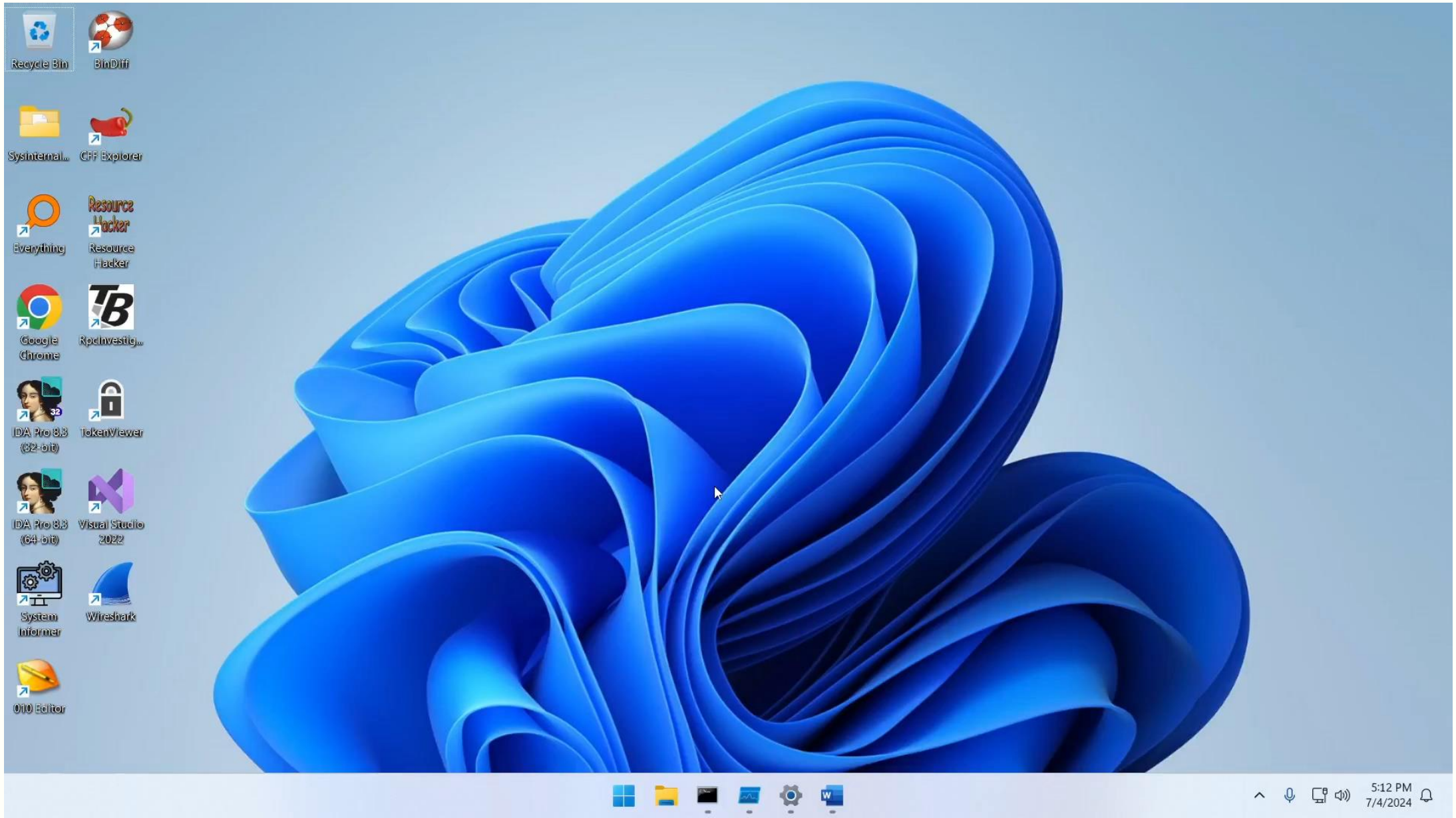
```
__int64 __fastcall CompatCacheRemoveExe(wchar_t *FilePath, HANDLE FileHandle)
{
  int status; // ebx
  CompatCacheAutoExe compatCache; // [rsp+20h] [rbp-E0h] BYREF
  struct _AHC_SERVICE_DATA serviceData; // [rsp+50h] [rbp-B0h] BYREF

  compatCache.m_Handle = 0i64;
  compatCache.m_ShouldCloseHandle = 0;
  memset(&compatCache.m_HeapBuffer, 0, 0x18);
  status = COMPAT_CACHE_AUTO_EXE::Init(&compatCache, FilePath, FileHandle);
  if ( status >= 0 )
  {
    memset_0(&serviceData, 0, sizeof(serviceData));
    serviceData.Remove.FileHandle = compatCache.m_Handle;
    serviceData.Remove.ExeSignature = compatCache.m_UnicodeStr;
    serviceData.Remove.InfoClass = 0xFFFF109F;
    status = (g_CompatCacheServiceApi)(ApphelpCacheServiceRemove, &serviceData);
  }
  COMPAT_CACHE_AUTO_EXE::~COMPAT_CACHE_AUTO_EXE(&compatCache);
  return status;
}
```

Attack #1 Flow

1. Write to disk the DLL that will be injected
2. Pick a scheduled task as a target ("Office Automatic Updates 2.0" for example)
3. Remove the shim cache for the file executed by the scheduled task
4. Request oplock for C:\windows\apppatch\sysmain.sdb
5. Run the scheduled task
6. The process will be created as SYSTEM in a suspended state
7. The scheduled tasks service will attempt to read sysmain.sdb, then enter a wait state
8. The callback from the oplock will be called in our process
9. Send an RPC to the OfficeClickToRun service to inject the DLL into the suspended process
10. Release the oplock so the scheduled tasks service will resume the process
11. The patched IAT will be used, and the DLL will be loaded into a SYSTEM process

Attack #1 Demo



Attack #1 Summary

- OfficeClickToRun.exe is not monitored by security products for code injection
- The injected process is spawned under the task scheduler
- Gained elevation to “NT AUTHORITY\SYSTEM”
- Calling `NtApphelpCacheControl` or requesting an OpLock is not suspicious

Agenda

1 Intro

Overview the shim framework
and known ways to abuse it

2 RPC in Office

Attack surface research on RPC servers
in MS Office

3 App Compatibility

Reverse engineer the mechanism to
renovate the attack

4 Conclusion

Detection methods and further
research

Digging into the Apphelp Cache

- What can we achieve with this API?
- What happens when the call to this API reaches the kernel?
- Down the rabbit hole we go



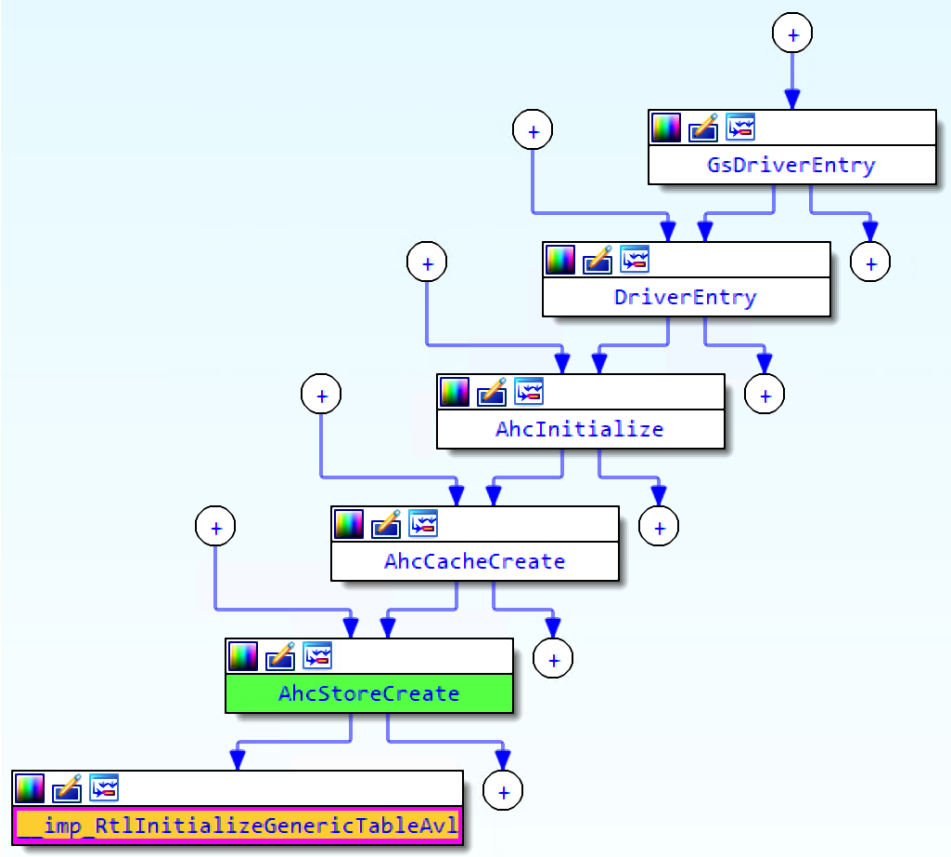
```
K 9   ntoskrnl.exe     NtCreateFile + 0x79
K 10  ntoskrnl.exe     KiSystemServiceCopyEnd + 0x25
K 11  ntoskrnl.exe     KiServiceLinkage
K 12  ahcache.sys      AslFileOpen + 0xa6
K 13  ahcache.sys      AhcCdbDetectServicing + 0x68
K 14  ahcache.sys      AhcCheckForChange + 0x7c
K 15  ahcache.sys      AhcCacheLookup + 0x83
K 16  ahcache.sys      AhcApiLookupAndWriteToProcess + 0xda
K 17  ahcache.sys      AhcDispatch + 0x2a7
K 18  ahcache.sys      AhcDriverDispatchDeviceControl + 0x41
K 19  ntoskrnl.exe     IofCallDriver + 0x55
K 20  ntoskrnl.exe     NtApphelpCacheControl + 0xf0
K 21  ntoskrnl.exe     KiSystemServiceCopyEnd + 0x25
U 22  ntdll.dll        NtApphelpCacheControl + 0x14
U 23  KERNEL32.DLL     CompatCacheLookupAndWriteToProcess + 0x237
U 24  KERNEL32.DLL     BasepQueryAppCompat + 0x2c2
U 25  KERNELBASE.dll   CreateProcessInternalW + 0x31bf
```

Digging into the Apphelp Cache

- When **`ahcache.sys`** is loaded, it reads the registry

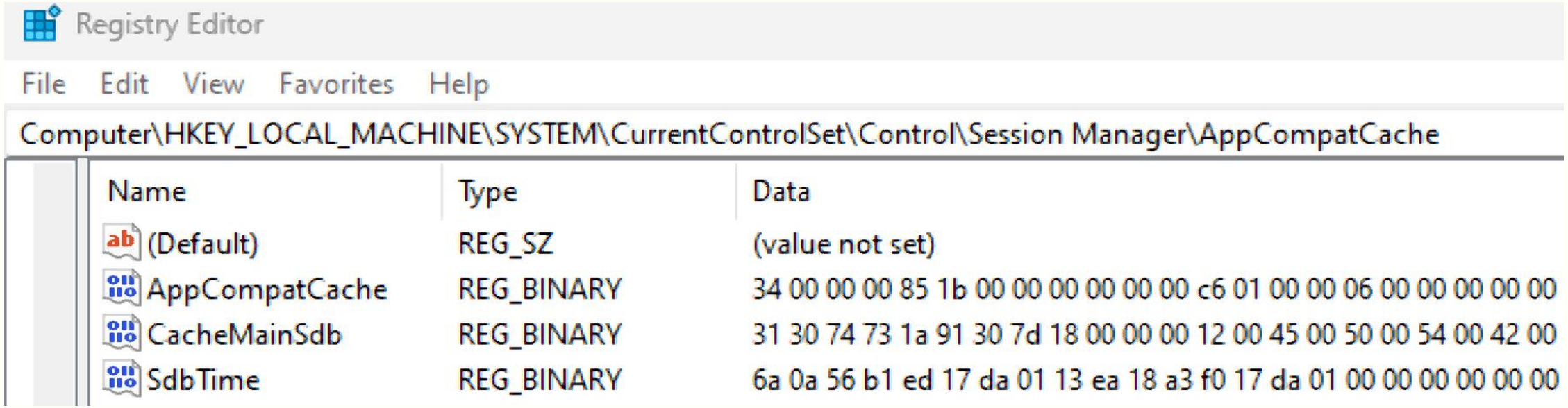
  `\Registry\MACHINE\System\CurrentControlSet\Control\Session Manager\AppCompatCache`

- These values contain binary data that the driver sorts into an AVL table.



Digging into the Apphelp Cache

- The table is dumped into the registry value on shutdown/restart.



Digging into the Apphelp Cache

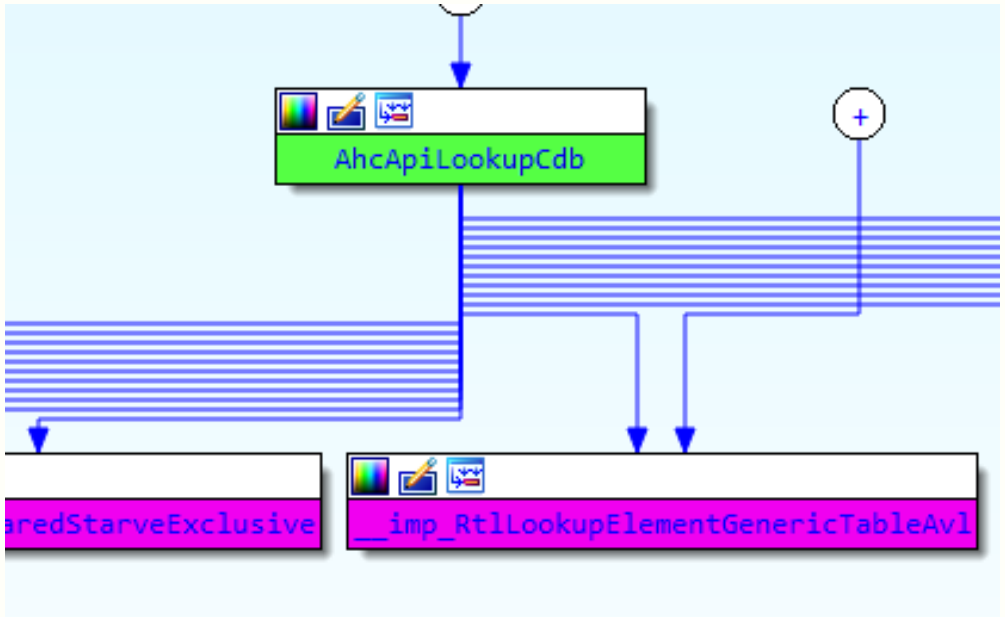
- Each "shimmable" entry in the table contains an **EXE TAG**
- Entries contain the size of data, data and the path of the executable
- Exe Tags point to the address inside sysmain.sdb

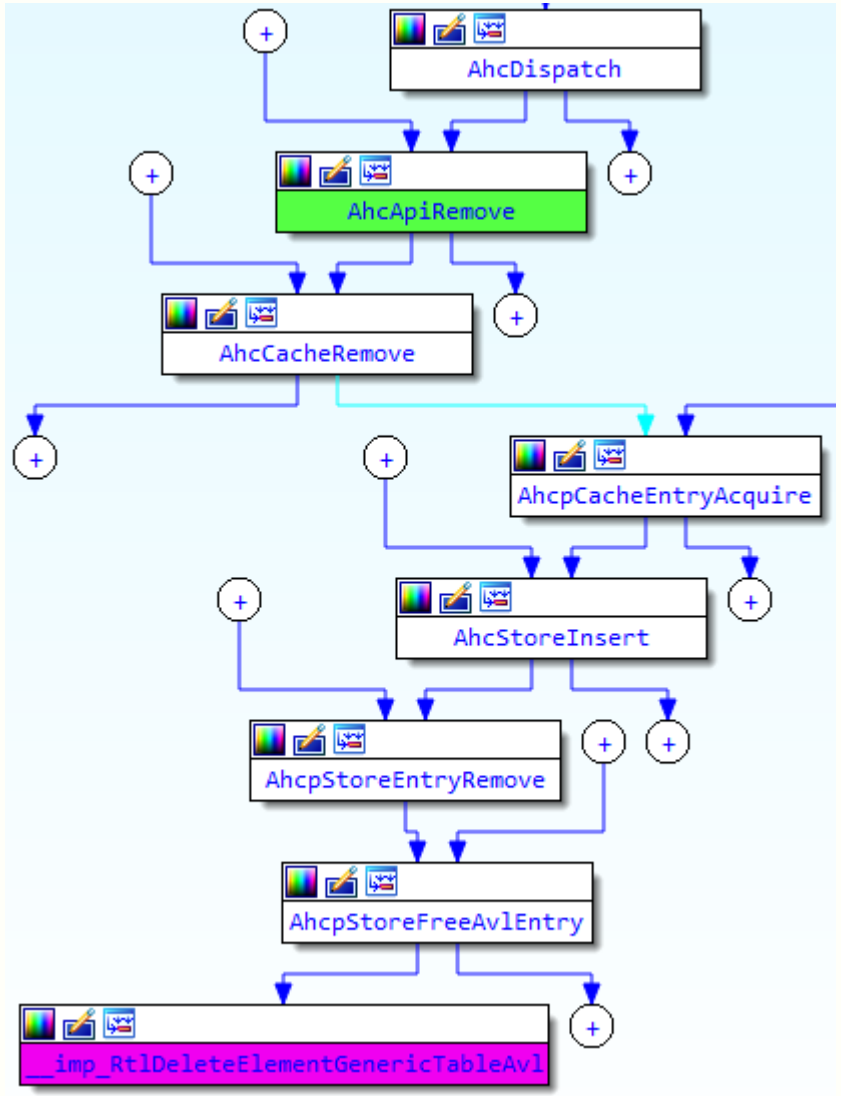
Path	Data Size (Bytes)	Data
C:\temp\VRCHAT.exe	544	0x0002000004000000000000000080000020000006486000001000000C80100009CCC2000000000…

```
20cc9c TAG 7007 - EXE
          20cca2 TAG 6001 - NAME: VRCHAT.EXE
          20cca8 TAG 6006 - APP_NAME: VRChat
          20ccae TAG 6005 - VENDOR: VRChat Inc.
          20ccb4 TAG 9004 - EXE_ID: {0C401607-0335-4E58-AF58-33E05474C578}
          20ccca TAG 9011 - APP_ID: {B300A362-4558-4336-A737-9917E38DEB23}
          20cce0 TAG 6036 - DEST_OS_GTE
          20cce6 TAG 7008 - MATCHING_FILE
                    20ccec TAG 6001 - NAME: *
          20ccf2 TAG 7009 - SHIM_REF
                    20ccf8 TAG 6001 - NAME: DXGICompat
                    20ccfe TAG 4004 - SHIM_TAGID: 201478 (0x31306)
                    20cd04 TAG 6008 - COMMAND_LINE:
          MaximizedWindowedFullscreen=1;DisableMaximizedWindowedSwapEffectTransition=1
```

Digging into the Apphelp Cache

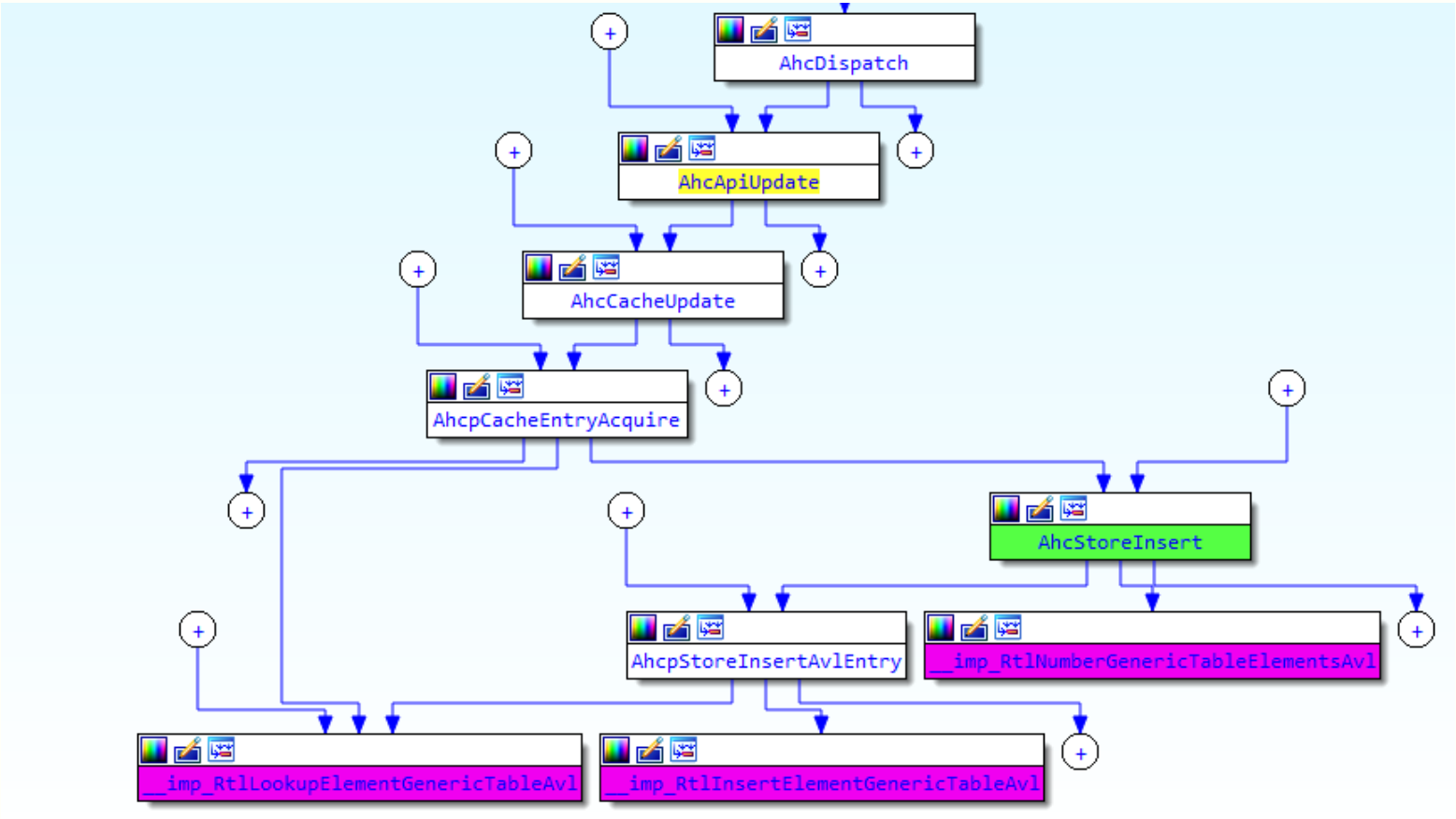
- `NtApphelpCacheControl` performs operations on the table
- `AhcApiLookupCdb`
- `AhcApiRemove`





Digging into the Apphelp Cache

- `AhcApiUpdate`



Digging into the Apphelp Cache

- We noticed a function that handles **`ApphelpCacheServiceInitProcessData`**
- **`AhcApiInitProcessData`** doesn't access the AVL table
- Instead, it writes into the memory of another process!

```
v5 = ObOpenObjectByPointer(v2, 0x200u, 0i64, 0x438u, 0i64, 0, &ProcessHandle);
if ( v5 < 0 )
  goto LABEL_47;
RegionSize = v4;
if ( ZwAllocateVirtualMemory(ProcessHandle, &BaseAddress, 0i64, &RegionSize, 0x1000u, 4u) < 0 || RegionSize < v4 )
{
  v5 = -1073741801;
  v11 = "Out of memory";
  v12 = 1309;
  goto LABEL_19;
}
LOBYTE(AccessMode) = 0;
v5 = MmCopyVirtualMemory(v30, v3, v2, BaseAddress, v4, AccessMode, &v29);
if ( v5 < 0 )
{
  AslLogCallPrintf(1, (unsigned int)"AhcApiInitProcessData", 1326, (unsigned int)"Error copying memory [%x]");
  goto LABEL_47;
}
peb->pShimData = BaseAddress;
if ( peb32 )
  peb32->pShimData = (unsigned int)BaseAddress;
```

Init Process Data

- `InitProcessData` operation uses the struct `_AHC_SERVICE_INIT_PROCESS_DATA`
- The Data points to another undocumented struct
- Combining reverse engineering and definitions from ReactOS:

```
struct _AHC_SERVICE_INIT_PROCESS_DATA
{
    HANDLE ProcessHandle;
    PVOID Data;
    ULONG DataSize;
}
```



```
typedef struct _SHIM_DATA
{
    DWORD Unknown1[130];
    DWORD Size;
    DWORD Magic;
    DWORD Unknown2[1004];
} SHIM_DATA, * PSHIM_DATA;
```

AhcApiInitProcessData

- Checks performed before writing the data:
  - The calling process runs as **`NT AUTHORITY\SYSTEM`** with **`SeTcbPrivilege`**
  - **`PEB.BitField`** of the target process has the flag **`IsPackagedProcess`** (0x10)
  - **`_EPROCESS.Protection`** of the target process is **`0x81`**
  - **`Data.Magic`** is **`0xAC0DEDAB`**
  - **`Data.Size`** is **`0x11C0`**
- **`PsGetProcessProtection`** fails
- No target was found

```
enum _PS_PROTECTED_TYPE {
    PsProtectedTypeNone,
    PsProtectedTypeProtectedLight = 1,
    PsProtectedTypeProtected,
    PsProtectedTypeMax,
}
```

```
enum _PS_PROTECTED_SIGNER {
    PsProtectedSignerNone,
    PsProtectedSignerAuthenticode,
    PsProtectedSignerCodeGen,
    PsProtectedSignerAntimalware,
    PsProtectedSignerLsa,
    PsProtectedSignerWindows,
    PsProtectedSignerWinTcb,
    PsProtectedSignerWinSystem,
    PsProtectedSignerApp = 8,
    PsProtectedSignerMax,
}
```

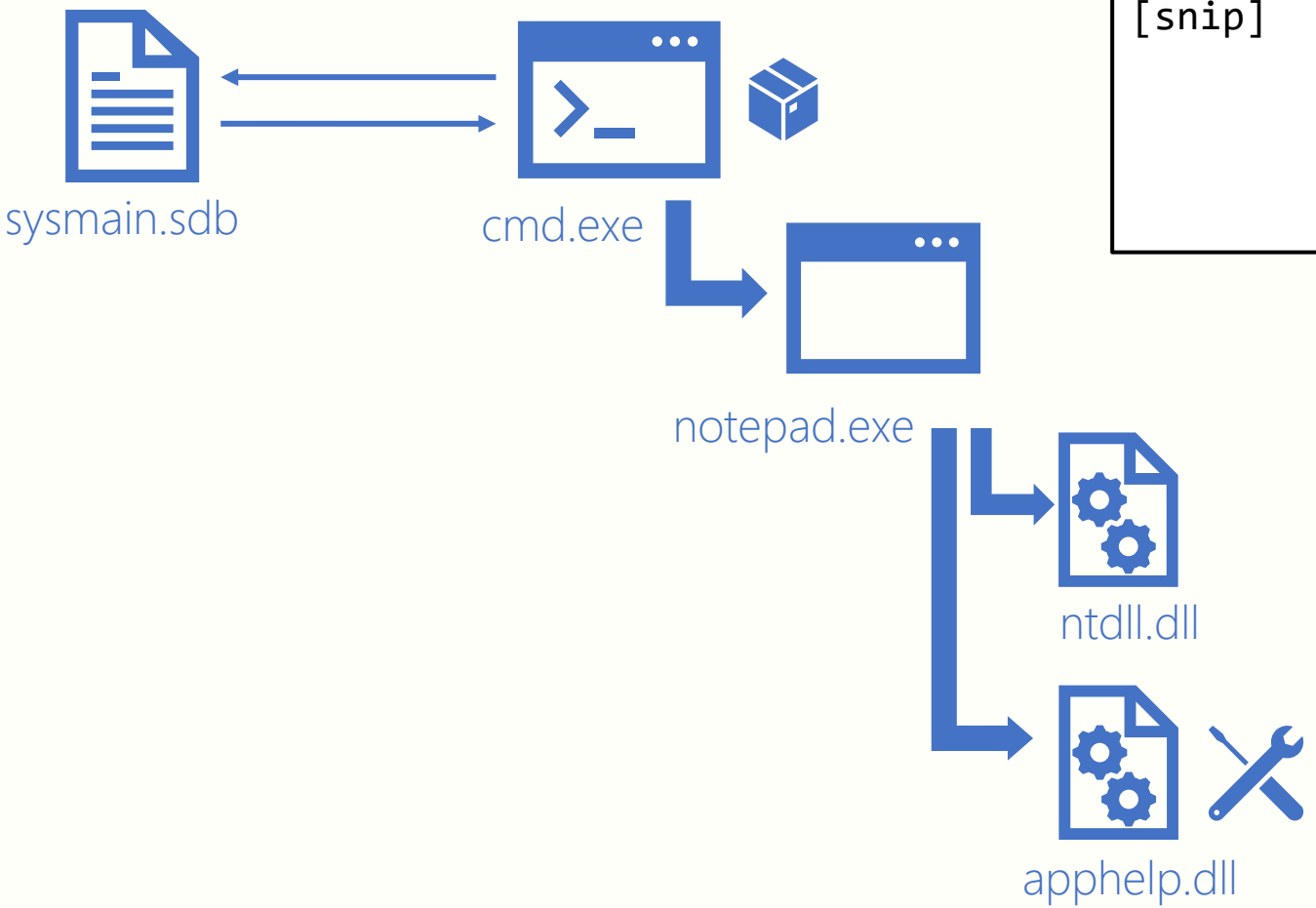
SHIM_DATA

- The magic value of `SHIM_DATA` was looked for in OS files
- It appears only in `ahcache.sys`, `kernel32.dll` and `apphelp.dll`.

Function	DLL	Read/Write	Exported
SdbUnpackAppCompatData	Apphelp.dll	Read	Yes
SdbPackAppCompatData	Apphelp.dll	Write	Yes
SdbGetAppCompatDataSize	kernel32.dll	Read	Yes
BaseReadAppCompatDataForProcessWorker	kernel32.dll	Read	Yes
BasepGetAppCompatData	kernel32.dll	Write	Yes
SbpGetShimData	kernel32.dll	Read	No

Compatibility Fixes Initialization

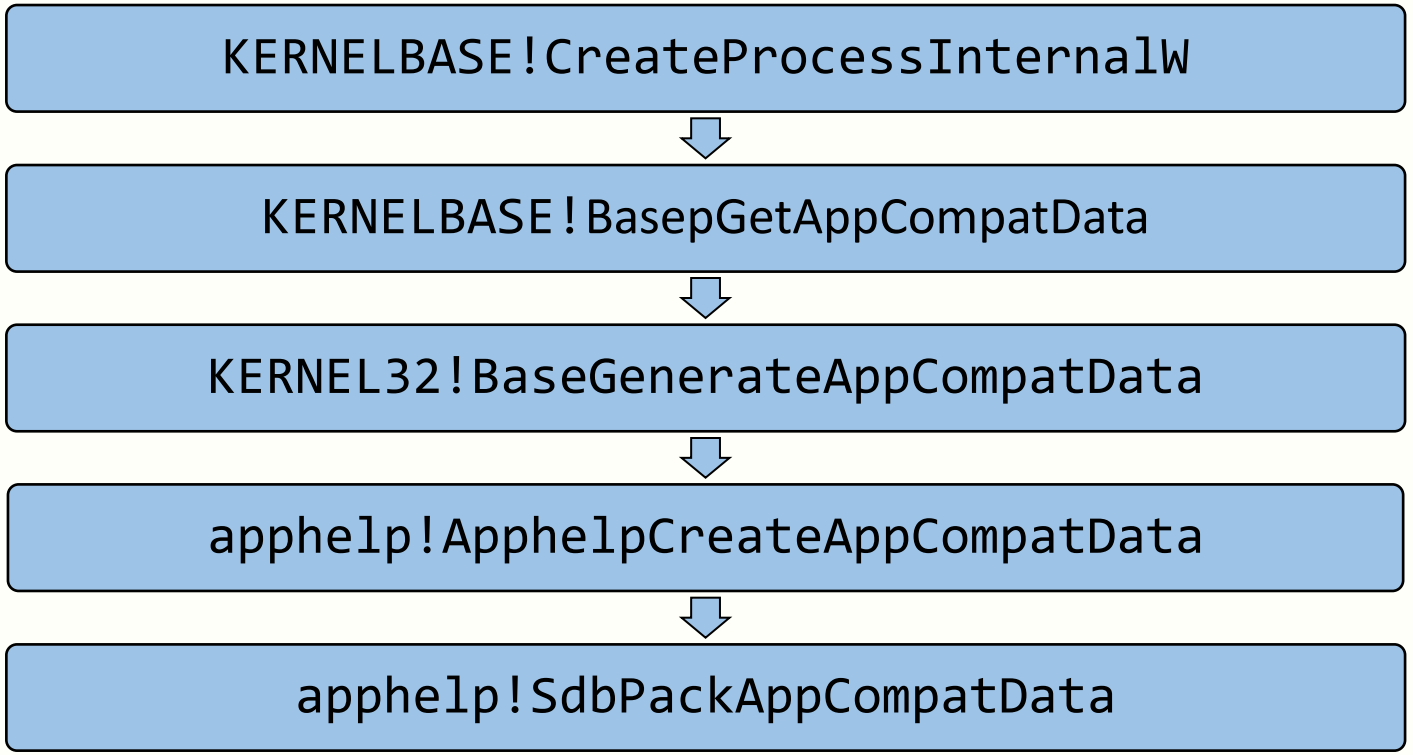
- When a new process is created as part of `kernelbase!CreateProcessInternalW`:




```
0:001> dt _PEB
[snip]
           +0x2c8 AppCompatFlags    : _ULARGE_INTEGER
           +0x2d0 AppCompatFlagsUser : _ULARGE_INTEGER
           +0x2d8 pShimData         : Ptr64 Void
           +0x2e0 AppCompatInfo     : Ptr64 Void
```

Creating SHIM_DATA

- After the shim cache is queried, the parent process builds the struct
- It is built as part of the call to `CreateProcess`

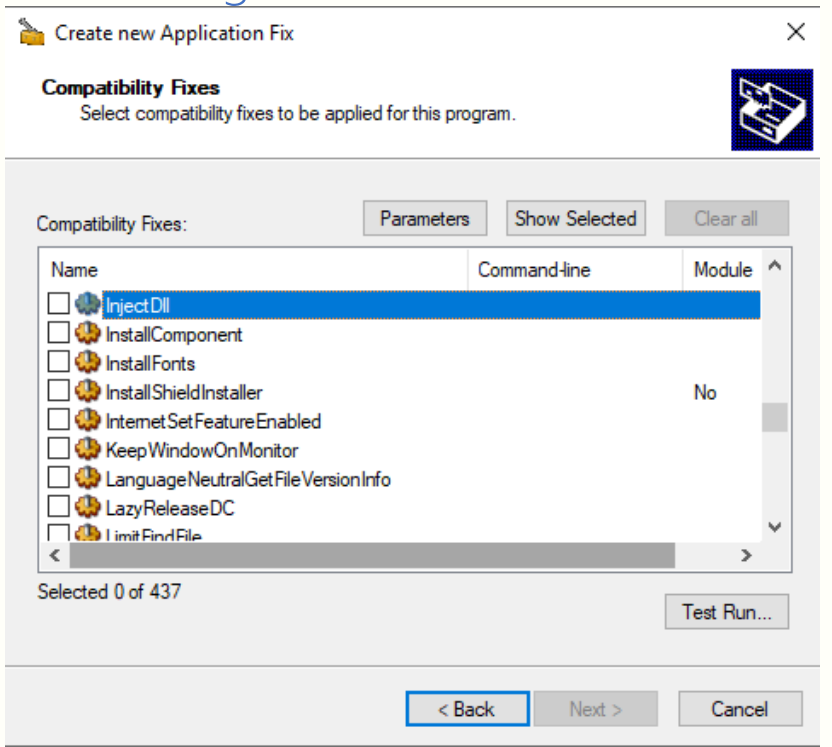


Getting to know pShimData

- What if we could manipulate **`apphelp.dll`** into applying malicious shim fixes?
- We need to discover what is written to **`PEB->pShimData`**
  - The "Compatibility Administrator" was used to install custom rules for shims
  - **`SHIM_DATA`** was dumped from process memory
  - Raw data was fuzzed and compared to other instances of custom and existing rules

```
0200h: 00 00 00 00 00 00 00 00 C0 11 00 00 AB ED 0D AC
0210h: 01 00 00 00 4C 01 00 00 DE C2 05 00 00 00 00 00
0220h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0230h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0240h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0250h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0260h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0270h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0280h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0290h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02A0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02B0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02C0h: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02D0h: 00 00 00 00 00 00 00 00 01 01 00 00 01 00 00 00
02E0h: 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
02F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0300h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0310h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0200h: 00 00 00 00 00 00 00 00 C0 11 00 00 AB ED 0D AC
0210h: 01 00 00 00 4C 01 00 00 46 03 00 30 00 00 00 00
0220h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0230h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0240h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0250h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0260h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0270h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0280h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0290h: 00 00 00 00 00 00 00 00 18 FA 03 00 00 00 00 00
02A0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02B0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02C0h: 01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
02D0h: 00 00 00 00 00 00 00 00 01 01 00 00 08 00 00 00
02E0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0300h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0310h: FF 08 82 C5 45 0E C5 43 8F CE 49 07 F8 2B A9 A7
```



pShimData

- This is the definition of the structure after reverse engineering:

```
typedef struct _SHIM_DATA
{
    DWORD Unknown1[130];
    DWORD Size;          // Always 0x11C0
    DWORD Magic;         // Always 0xAC0DEDAB
    DWORD UpBit;         // Always 0x1
    DWORD ExeType;       // 0x8664 (64-bit), 0x14c (32-bit)
    DWORD ExeTag;        // shim "id", pointing to the address in the sdb
    DWORD Unknown2[20];
    DWORD Tag2;          // exe or layer tag (optional)
    DWORD Tag3;          // exe or layer tag (optional)
    DWORD Unknown3[19];
    DWORD ExeCount;      // number of shims to apply
    DWORD LayerCount;    // number of layers to apply
    DWORD Unknown4[958];
} SHIM_DATA, * PSHIM_DATA;
```

Preparations

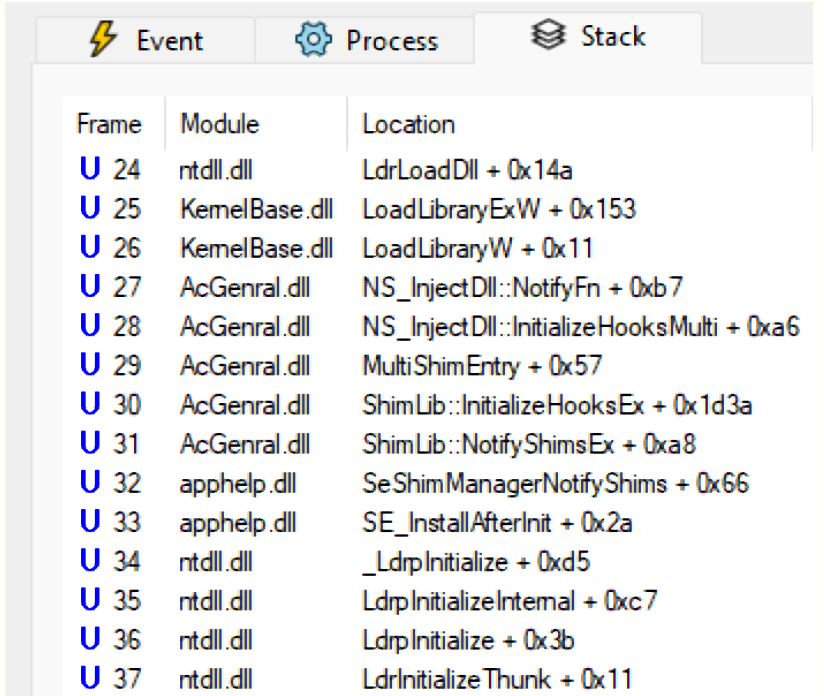
- We aim to avoid registering a custom SDB file to evade detection by security products
- We need to find an existing "Inject DLL" fix entry in **`sysmain.sdb`**

```
4ed54 TAG 7007 - EXE
        4ed5a TAG 6001 - NAME: GLJ*.tmp
        4ed60 TAG 600b - WILDCARD_NAME
        4ed66 TAG 6006 - APP_NAME: AOL Instant Messenger
        4ed6c TAG 6005 - VENDOR: AOL
        4ed72 TAG 9004 - EXE_ID: {F83A0FCD-6D48-4714-8A38-D06D376AA7A0}
        4ed88 TAG 9011 - APP_ID: {D1591404-7C1C-4A8E-9395-083889FF63BB}
        4ed9e TAG 4021 - RUNTIME_PLATFORM
        4eda4 TAG 7008 - MATCHING_FILE
                4edaa TAG 6001 - NAME: *
                4edb0 TAG 4001 - SIZE
                4edb6 TAG 4003 - CHECKSUM: 1229221402 (0x49446e1a)
                4edbc TAG 6024 - EXPORT_NAME
        4edc2 TAG 7009 - SHIM_REF
                4edc8 TAG 6001 - NAME: InjectDll
                4edce TAG 4004 - SHIM_TAGID: 177942 (0x2b716)
                4edd4 TAG 6008 - COMMAND_LINE: RTvideo.dll
```

Limitations

1. The injected process must be launched as suspended
   - `apphelp!SE_InstallAfterInit` is preventing shim data processing
2. The injected process cannot be a system file
   - `ShimLib::IsSystemFile` skips injection for files :
     - ❑ Under C:\Windows\WinSXS
     - ❑ Owned by "Trusted Installer"
3. This shim fix is unavailable for 64-bit processes
   - 64-bit version of `AcGenral.dll` lacks the `NS_InjectDll` class

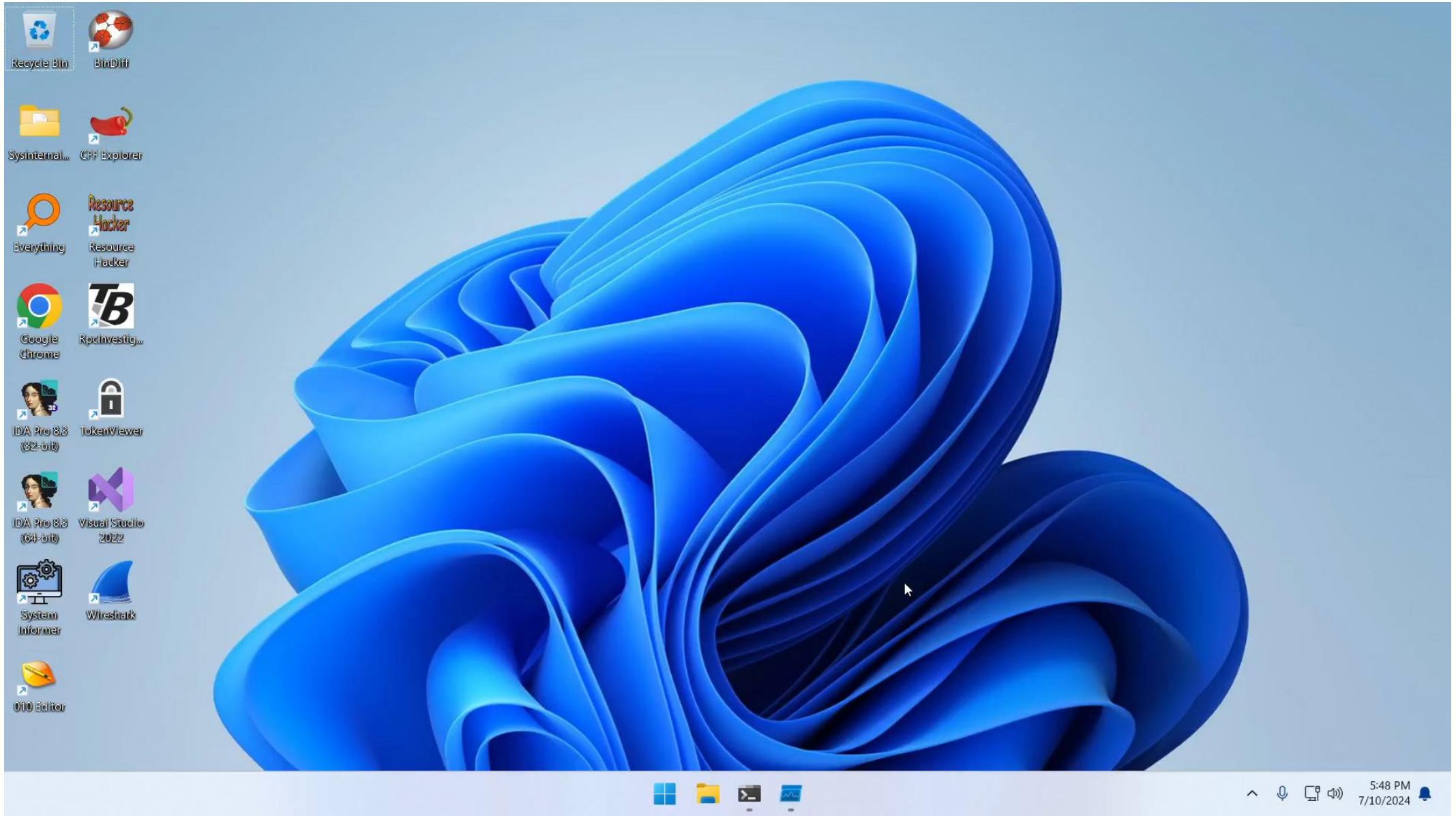
- And so, `MicrosoftEdgeUpdate.exe` will be the target of the attack
- It is found in: `C:\Program Files (x86)\Microsoft\EdgeUpdate`



Attack #2 Flow

1. Write a DLL to disk named `RTvideo.dll`
2. Launch a suspended process in the same directory
3. Build a `SHIM_DATA` structure and set the `ExeTag` to be `0x4ed54`
4. Write the structure to the memory of the child process
5. Point `PEB->pShimData` to the structure
6. Resume the child process
7. apphelp.dll will apply the `InjectDll` fix and our DLL will be loaded

Attack #2 Demo



Attack #2 Summary

- Malicious shims are a known attack vector that is detected by many security products
- By reverse engineering this infrastructure we were able to renovate it
- We found a novel method to make this attack fileless AND registry-less
- We applied a malicious shim on a process without registering an SDB file on the system
- We bypassed EDR monitoring
- We chose the "`InjectDll`" shim, but this attack can be adapted to other malicious shim fixes

Agenda

1 Intro

Overview the shim framework and known ways to abuse it

2 RPC in Office

Attack surface research on RPC servers in MS Office

3 App Compatibility

Reverse engineer the mechanism to renovate the attack

4 Conclusion

Detection methods and further research

Detection

- First Attack:
  - RPC to ClickToRunSvc with unusual DLL paths
  - Requesting OpLock for **`sysmain.sdb`**
  - ClickToRunSvc writing to the memory of a process running as "**`NT AUTHORITY\SYSTEM`**"
- Second Attack:
  - Writing DLL that appears in **`sysmain.sdb`** to an unusual directory

Further Research - App Compatibility

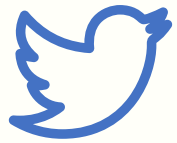
- Find target for `AhcApiInitProcessData`
- Reverse engineer `SHIM_DATA` to find more ways to manipulate apphelp.dll
- Poison cache to cause another process to apply a malicious shim

Conclusion

- Methodology for attack surface research was presented
- Several components were manipulated into a single attack
- The oplock and compatibility mechanisms
- This research led to new stealthy injection and privilege escalation techniques
- Both attacks bypass EDR monitoring
- Reverse-engineered an undocumented API and structures
- Renovated a known malicious technique to make it fileless and registry-less
- We invite the community to continue the research and explore shared leads

Thank you

Contact us:



@RonB_Y
@DavidShandalov



https://www.linkedin.com/in/ron-by
https://www.linkedin.com/in/david-shan



GitHub repo: https://github.com/deepinstinct/ShimMe

