Example of Windows Warbird Encryption/Decryption

downwithup.github.io/blog/post/2023/04/23/post9.html

> Everything in this post was done on a Windows 10 22H2 machine. Kernel version was: 10.0.19041.2486

Introduction

Microsoft Warbird is an undocumented encryption technology generally used for things relating to software licensing (DRM) and security mechanisms. There has been some, but not much, previous open source research. Some links which provide further insight:

- https://github.com/KiFilterFiberContext/warbird-obfuscator
- https://github.com/KiFilterFiberContext/microsoft-warbird/

The Warbird technology is appears to be designed to be integrated at compile time, and could function either as an obfuscation approach on the existing code, or as some type of "enclave" block encryptor. This second approach is what this post will dive into.

SystemControlFlowTransition

There is a semi-undocumented system information class for `NtQuerySystemInformation` called `SystemControlFlowTransition` (0xB9) which when called ends up in the `WbDispatchOperation` function. Placing a breakpoint on this function will show that the `sppsvc.exe` process periodically calls this. More on this later. `WbDispatchOperation` will branch into several different functions depending on the `operation` value passed when calling `NtQuerySystemInformation`. The struct looks something like this:

```
typedef struct _WB_OPERATION
{
        ULONG Operation;
        PVOID Buffer;
    ... (operation dependent data)
} WB_OPERATION, *PWB_OPERATION;
```

These are the operations:

- 1 = WbDecryptEncryptionSegment
- 2 = WbReEncryptEncryptionSegment
- 3 = WbHeapExecuteCall
- 4 = *non symbol name function*
- 5 = *non symbol name function.*
- 6 = *same as case 5*

- 7 = WbRemoveWarbirdProcess
- 8 = WbProcessStartup
- 9 = WbProcessModuleUnload
  Each one of these operations has some type of unique operation dependent data attached to the initial struct. Reversing the `sppsvc.exe` can give us hints on how these structures *should* be formatted and how they are called. The decrypt and re-encrypt steps can occur multiple times. The rough pseudocode based on `sppsvc.exe` for calling `WbProcessStartup` looks like this:

```
SystemInfo[0] = 8;                      // Operation (WbProcessStartup)

SystemInfo[1] = &buffer;
NtQuerySystemInformation(0xB9, SystemInfo, 0x10, NULL);
```

Where `buffer`:

```
{
    ULONG: 0,
    ULONG: 0x64,
    ULONG64: 0,
    ULONG: 0
}
```

The name `WbProcessStartup` seems to suggest that sometype this call does some form of initialization which is required before decrypting/reencrypting data. However, this does not appear to be the case, and the calls to decrypt/reencrypt seem to work without. The rough pseudocode based on `sppsvc.exe` for calling `WbDecyptEncryptionSegment` looks like this:

```
SystemInfo[0] = 1;                      // Operation

SystemInfo[1] = WarbirdPayload;         // At this point it is encrypted

SystemInfo[2] = PEBaseAddress;          // Base Address of the PE

SystemInfo[3] = 0x140000000;            // Image Base

SystemInfo[4] = UnknownLong64;          // Possibly something relating to encryption

SystemInfo[5] = 0x2;                    // Unknown flags

result = NtQuerySystemInformation(0xB9, SystemInfo, 0x30u, NULL);
```

It’s important to note that the `WarbirdPayload` is actually embedded in the `sppsvc.exe` binary in a section named `?g_Encry`. There are multiple of these sections.

Payload Format

For decryption (`WbDecyptEncryptionSegment`) the payload is in the format of `WB_PAYLOAD` structure.

```
typedef struct _WB_SEGMENT
{
        ULONG Flags;
        ULONG RVA;
        ULONG Length;
} WB_SEGMENT, *PWB_SEGMENT;

typedef struct _FEISTEL_ROUND
{
        ULONG One;
        ULONG Two;
        ULONG Three;
        ULONG Four;
} FEISTEL_ROUND, *PFEISTEL_ROUND;

typedef struct _WB_PAYLOAD {
        BYTE Hash[0x20];        // SHA 256 hash of the payload sha256(payload size - 0x20)

        ULONG TotalSize;                // Total size (includes all segments)

        ULONG Reserved;                 // Set to 0

        ULONG PayloadRVA;               // Offset between start of payload struct and
actual start of the data passed (WarbirdPayload) in the NtQuerySystemInformation call

        ULONG SecondStageRVA;   // Offset between start of second stage struct and actual
start of the data passed (WarbirdPayload) in the NtQuerySystemInformation call

        ULONG SecondStageSize;  // Size of the UnknownData in DWORDs

        ULONG UnknownLong;              // Looks like this is reserved. Must be 0?

        ULONG64 ImageBase;              // PE image base

        BYTE Unknown2[0x8];             // Looks like this is reserved. Must be 0?

        ULONG64 FeistelKey;
        FEISTEL_ROUND Rounds[10];
        ULONG SegmentCount;             // Number of segments

        WB_SEGMENT Segments[1]; // Segment struct(s)

} WB_PAYLOAD, * PWB_PAYLOAD;
```

The most important field is the `Segments`, an array of `WB_SEGMENT` structures. These point (using RVA) to the encrypted blocks of code to be decrypted. The flags field in the `WB_SEGMENT` specify what protection the segment should be decrypted as. If any value is present, it is a `PAGE_EXECUTE_READ` else it is `PAGE_READONLY`.

How to Encrypt

As you may have noticed in the supported operations values, and from the description of the `sppsvc.exe` usage, there is no encrypt. This is most likely because this API is intended to be used only after a binary is compiled with the Warbird encrypted chunks. To get around this, you can use the `WbReEncryptEncryptionSegment` functionality to first decrypt some random data, replace that data with the bytes we want to encrypt. Then, reencrypt this same memory. If you then save this strucutre (the segment bytes as well as the payload structure) you can then have memory that when restored, can simply be decrypted.

The Mitigation

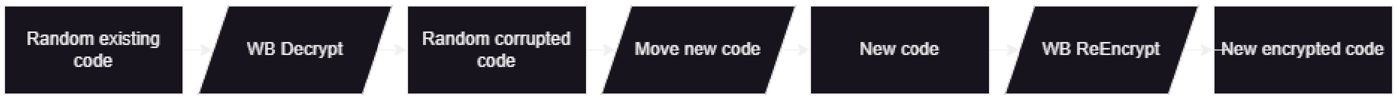
Note that `sppsvc.exe` is a Windows signed binary. This brings us to a problem. In Alex Ionescu's talk he explained that part of the patch Microsoft made to fix the bug he found was only allow decryption of payloads that were signed by the Windows team at Microsoft. The kernel does this by calling `ZwQueryVirtualMemory` with the `MemoryImageInformation` class on the memory passed as the payload. Process Hacker's NT headers have the structure for this undocumented memory class. The `ImageFlags` is then compared to ensure the memory was backed with the appropriate signature.

The Bypass

This however, is not a perfect mitigation as at runtime memory which has been backed by a PE with specific signatures can be modified simply by changing the existing virtual memory protections (RX to RW or RWX).

Putting it all Together

Here is a simplified view of how this whole process will work. The "code" resides within the address space of a signed image.



PoC Code

This PoC will simply follow the steps above. In summary, this will load a signed DLL as the scratch space, then decrypting, writing code, reencrypting, and finally decrypting again.

Posted on Apr 23, 2023