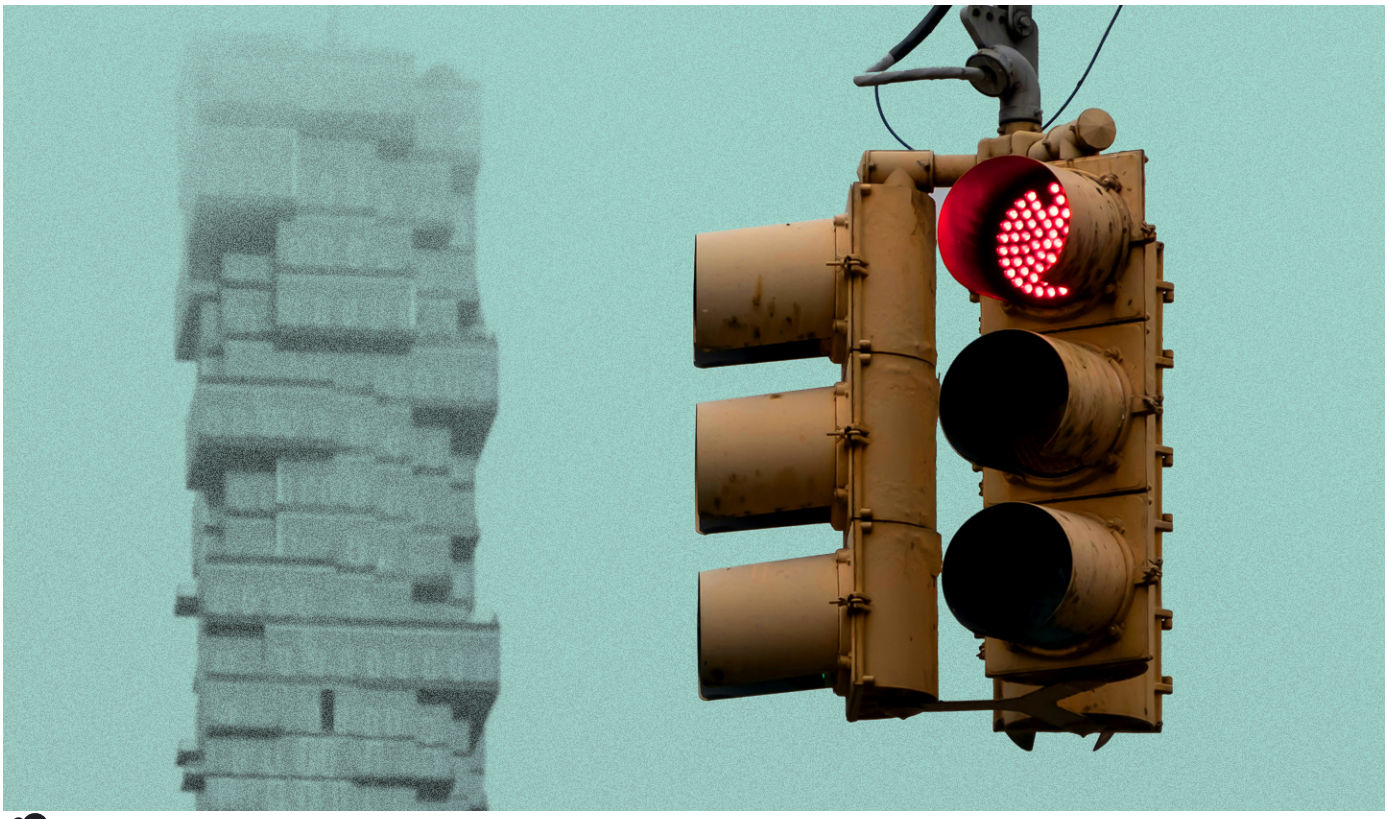
Investigating a Mysteriously Malformed Authenticode Signature



4 September 2025•Elastic Security Labs

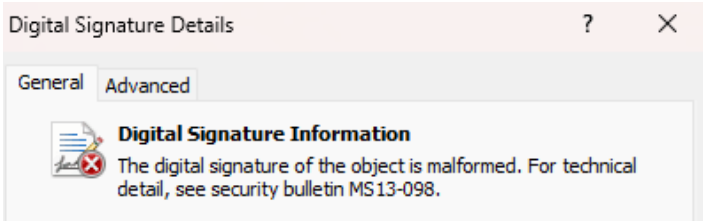
Uncovering the hidden heuristics behind malformed authenticode signatures

16 min readSecurity operations

Introduction

Elastic Security Labs recently encountered a signature validation issue with one of our Windows binaries. The executable was signed using `signtool.exe` as part of our standard continuous integration (CI) process, but on this occasion, the output file failed signature validation with the following error message:

> The digital signature of the object is malformed. For technical detail, see security bulletin MS13-098.



MS13-098 error

The documentation for MS13-098 is vague, but it describes a potential vulnerability related to malformed Authenticode signatures. Nothing obvious had changed on our end that might explain this new error, so we needed to investigate the cause and resolve the issue.

While we identified that this issue was affecting one of our signed Windows binaries, it could impact any binary. We are publishing this research as a reference for anyone else who may encounter the same problem in the future.

Diagnosis

To investigate further, we created a basic test program that called the Windows `WinVerifyTrust` function against the problematic executable to manually validate the signature. This revealed that it was failing with the error code `TRUST_E_MALFORMED_SIGNATURE`.

`WinVerifyTrust` is a complex function, but after attaching a debugger, we discovered that the error code was being set at the following point:

```
dwReserved1 = psSipSubjectInfo->dwReserved1;

if(!dwReserved1)

    goto LABEL_58;

v40 = I_GetRelaxedMarkerCheckFlags(a1, v22, (unsigned int *)&pvData);

if(v40 < 0)

    break;

if(!pvData)

    v42 = 0x80096011;    // TRUST_E_MALFORMED_SIGNATURE
```

As shown above, if `psSipSubjectInfo->dwReserved1` is not `0`, the code calls `I_GetRelaxedMarkerCheckFlags`. If this function returns no data, the code sets the `TRUST_E_MALFORMED_SIGNATURE` error and exits.

When stepping through the code with our problematic binary, we saw that `dwReserved1` was indeed set to `1`. Running the same test against a correctly signed binary, this value was always `0`, which skips the call to `I_GetRelaxedMarkerCheckFlags`.

Looking into `I_GetRelaxedMarkerCheckFlags`, we saw that it simply checks for the presence of a specific attribute: `1.3.6.1.4.1.311.2.6.1`. A quick online search turned up very little other than the fact that this object

identifier (OID) is labeled as `SpcRelaxedPEMarkerCheck`.

```
__int64 __fastcall I_GetRelaxedMarkerCheckFlags(struct _CRYPT_PROVIDER_DATA *a1,
DWORD a2, unsigned int *a3)

{

    unsigned int v4; // ebx

    CRYPT_PROVIDER_SGNR *ProvSignerFromChain; // rax

    PCRYPT_ATTRIBUTE Attribute; // rax

    signed int LastError; // eax

    DWORD pcbStructInfo; // [rsp+60h] [rbp+18h] BYREF


    pcbStructInfo = 4;

    v4 = 0;

    *a3 = 0;

    ProvSignerFromChain = WTHelperGetProvSignerFromChain(a1, a2, 0, 0);

    if(ProvSignerFromChain)

    {

        Attribute = CertFindAttribute(

            "1.3.6.1.4.1.311.2.6.1",

            ProvSignerFromChain->psSigner->AuthAttrs.cAttr,

            ProvSignerFromChain->psSigner->AuthAttrs.rgAttr);

        if(Attribute)

        {

            if(!CryptDecodeObject(

                a1->dwEncoding,

                (LPCSTR)0x1B,

                Attribute->rgValue->pbData,

                Attribute->rgValue->cbData,

                0,
```

```
                a3,
                &pcbStructInfo))
            {
                return HRESULT_FROM_WIN32(GetLastError());
            }
        }
    }

    return v4;
}
```

Our binary did not have this attribute, which caused the function to return no data and triggered the error. The function names reminded us of an optional parameter that we had previously seen in `signtool.exe`:

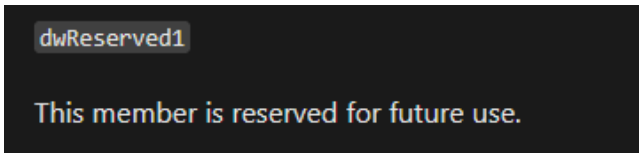
> `/rmc` - Specifies signing a PE file with the relaxed marker check semantic. The flag is ignored for non-PE files. During verification, certain authenticated sections of the signature will bypass invalid PE markers check. This option should only be used after careful consideration and reviewing the details of MSRC case MS12-024 to ensure that no vulnerabilities are introduced.

Based on our analysis, we suspected that re-signing the executable with the "relaxed marker check" flag (`/rmc`), and as expected, the signature was now valid.

Root cause analysis

While the workaround above resolved our immediate problem, it clearly wasn't the root cause. We needed to investigate further to understand why the internal `dwReserved1` flag was set in the first place.

This field is part of the `SIP_SUBJECTINFO` structure, which is documented on MSDN - but unfortunately, it didn't help much in this case:



SIP_SUBJECTINFO structure comment

To find where this field was being set, we worked backwards and identified a point where `dwReserved1` was still `0` - i.e., before the flag had been set. We placed a hardware breakpoint (on write) on the `dwReserved1` field and resumed execution. The breakpoint was hit in the `SIPObjectPE_::GetMessageFromFile` function:

```
__int64 __fastcall SIPObjectPE_::GetMessageFromFile(
```

```
    SIPObjectPE_ *this,

    struct SIP_SUBJECTINFO_ *a2,

    struct _WIN_CERTIFICATE *a3,

    unsigned int a4,

    unsigned int *a5)

{

    __int64 v5; // rcx

    __int64 result; // rax

    DWORD v8; // [rsp+40h] [rbp+8h] BYREF


    v5 = *((_QWORD*)this + 1);

    v8 = 0;

    result = ImageGetCertificateDataEx(v5, a4, a3, a5, &v8);

    if((_DWORD)result)

        a2->dwReserved1 = v8;


    return result;

}
```

This function calls the `ImageGetCertificateDataEx` API which is exported by `imagehlp.dll`. The value returned by the fifth parameter of this function is stored in `dwReserved1`. This value ultimately determines whether the PE is considered "malformed" in the manner we have been observing.

Unfortunately, `ImageGetCertificateDataEx` is undocumented on MSDN. However, an earlier variant, `ImageGetCertificateData`, is documented:

```
BOOL IMAGEAPI ImageGetCertificateData(

  [in]      HANDLE            FileHandle,

  [in]      DWORD             CertificateIndex,

  [out]     LPWIN_CERTIFICATE Certificate,

  [in, out] PDWORD            RequiredLength
```

```
);
```

This function extracts the contents of the `IMAGE_DIRECTORY_ENTRY_SECURITY` directory from the PE headers. Manual analysis of the `ImageGetCertificateDataEx` function showed that the first four parameters match those of `ImageGetCertificateData`, but with one additional output parameter at the end.

We wrote a simple test program that allows us to call this function and perform checks against the unknown fifth parameter:

```
#include <stdio.h>

#include <windows.h>

#include <imagehlp.h>


int main()

{

    HANDLE hFile = NULL;

    DWORD dwCertLength = 0;

    WIN_CERTIFICATE *pCertData = NULL;

    DWORD dwUnknown = 0;

    BOOL (WINAPI *pImageGetCertificateDataEx)(HANDLE FileHandle, DWORD
CertificateIndex, LPWIN_CERTIFICATE Certificate, PDWORD RequiredLength, DWORD
*pdwUnknown);



    // open target executable

    hFile = CreateFileA("C:\\users\\matthew\\sample-executable.exe", GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

    if(hFile == INVALID_HANDLE_VALUE)

    {

        printf("Failed to open input file\n");

        return 1;

    }



    // locate ImageGetCertificateDataEx export in imagehlp.dll
```

```
    pImageGetCertificateDataEx = (BOOL(WINAPI*)
(HANDLE,DWORD,LPWIN_CERTIFICATE,PDWORD,DWORD*))GetProcAddress(LoadLibraryA("imagehlp.dll"
"ImageGetCertificateDataEx");

    if(pImageGetCertificateDataEx == NULL)

    {

        printf("Failed to locate ImageGetCertificateDataEx\n");

        return 1;

    }


    // get required length

    dwCertLength = 0;

    if(pImageGetCertificateDataEx(hFile, 0, NULL, &dwCertLength, &dwUnknown) == 0)

    {

        if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)

        {

            printf("ImageGetCertificateDataEx error (1)\n");

            return 1;

        }

    }


    // allocate data

    printf("Allocating %u bytes for certificate...\n", dwCertLength);

    pCertData = (WIN_CERTIFICATE*)malloc(dwCertLength);

    if(pCertData == NULL)

    {

        printf("Failed to allocate memory\n");

        return 1;

    }
```

```
    // read certificate data and dwUnknown flag
    if(pImageGetCertificateDataEx(hFile, 0, pCertData, &dwCertLength, &dwUnknown) ==
0)
    {
        printf("ImageGetCertificateDataEx error (2)\n");
        return 1;
    }

    printf("Finished - dwUnknown: %u\n", dwUnknown);

    return 0;
}
```

Running this against a variety of executables confirmed our expectations: the unknown return value was `1` for our "broken" executable, and `0` for correctly signed binaries. This confirmed that the issue originated somewhere within the `ImageGetCertificateDataEx` function.

Further analysis of this function revealed that the unknown flag is being set by another internal function: `IsBufferCleanOfInvalidMarkers`.

```
...
if(!IsBufferCleanOfInvalidMarkers(v25, v15, pdwUnknown))
{
    LastError = GetLastError();
    if(!pdwUnknown)
        goto LABEL_34;
}
...
```

After cleaning up the `IsBufferCleanOfInvalidMarkers` function, we observed the following:

```
DWORD IsBufferCleanOfInvalidMarkers(BYTE *pData, DWORD dwLength, DWORD
*pdwInvalidMarkerFound)
```

```
{
    if(!_InterlockedCompareExchange64(&global_InvalidMarkerList, 0, 0))
        LoadInvalidMarkers();

    if(!RabinKarpFindPatternInBuffer(pData, dwLength, pdwInvalidMarkerFound))
        return 1;

    SetLastError(0x80096011); // TRUST_E_MALFORMED_SIGNATURE

    return 0;
}
```

This function loads a global list of "invalid markers" using `LoadInvalidMarkers`, if they are not already loaded. `imagehlp.dll` contains a hardcoded default list of markers, but also checks the registry for a user-defined list at the following path:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Cryptography\Wintrust\Config\PECertInvalidMarkers`

This registry value does not appear to exist by default.

The function then performs a search across the entire PE signature data, looking for any of these markers. If a match is found, `pdwInvalidMarkerFound` is set to `1`, which maps directly to the `psSipSubjectInfo->dwReserved1` value mentioned earlier.

Dumping the invalid markers

The markers are stored in an undocumented structure inside `imagehlp.dll`. After reverse-engineering the `RabinKarpFindPatternInBuffer` function noted above, we wrote a small tool to dump the entire list of markers:

```
#include <stdio.h>
#include <windows.h>

int main()
{
    HMODULE hModule = LoadLibraryA("imagehlp.dll");
```

```
// hardcoded address - imagehlp.dll version:
// 509ef25f9bac59ebf1c19ec141cb882e5c1a8cb61ac74a10a9f2bd43ed1f0585
BYTE *pInvalidMarkerData = (BYTE*)hModule + 0xC4D8;

BYTE *pEntryList = (BYTE*)*(DWORD64*)(pInvalidMarkerData + 20);
DWORD dwEntryCount = *(DWORD*)pInvalidMarkerData;
for(DWORD i = 0; i < dwEntryCount; i++)
{
    BYTE *pCurrEntry = pEntryList + (i * 18);
    BYTE bLength = *(BYTE*)(pCurrEntry + 9);
    BYTE *pString = (BYTE*)*(DWORD64*)(pCurrEntry + 10);
    for(DWORD ii = 0; ii < bLength; ii++)
    {
        if(isprint(pString[ii]))
        {
            // printable character
            printf("%c", pString[ii]);
        }
        else
        {
            // non-printable character
            printf("\\x%02X", pString[ii]);
        }
    }
    printf("\n");
}

return 0;
```

```
}
```

This produced the following results:

```
PK\x01\x02
PK\x05\x06
PK\x03\x04
PK\x07\x08
Rar!\x1A\x07\x00
z\xBC\xAF'\x1C
**ACE**
!<arch>\x0A
MSCF\x00\x00\x00\x00
\xEF\xBE\xAD\xDENull
Initializing Wise Installation Wizard
zlb\x1A
KGB_arch
KGB2\x00
KGB2\x01
ENC\x00
disk%i.pak
>-\x1C\x0BxV4\x12
ISc(
Smart Install Maker
\xAE\x01NanoZip
;!@Install@
EGGA
ArC\x01
StuffIt!
-sqx-
PK\x09\x0A
"\x0B\x01\x0B
-lh0-
-lh1-
-lh2-
-lh3-
-lh4-
-lh5-
-lh6-
-lh7-
-lh8-
-lh9-
-lha-
-lhb-
-lhc-
-lhd-
```

```
-lhe-
-lzs-
-lz2-
-lz3-
-lz4-
-lz5-
-lz7-
-lz8-
<#$@@$#>
```

As expected, this appears to be a list of magic values pertaining to old installers and compressed archive formats. This aligns with the description of MS13-098, which hints towards certain installers being affected.

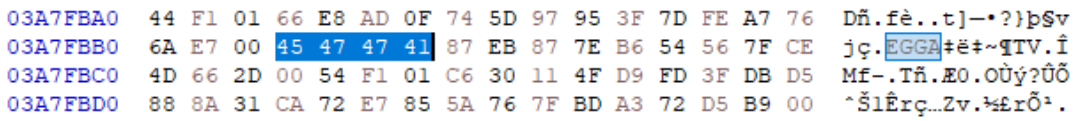
We suspected this was related to self-extracting executables. If an executable reads itself from disk and scans its own data for an embedded archive (e.g., a ZIP file), an attacker could potentially append malicious data to the signature section without invalidating the signature - since signature data cannot hash itself. This could potentially cause the vulnerable executable to locate the malicious data before the original data, especially if it scans backwards from the end of the file.

We later found an old RECon talk from 2012 by Igor Glücksmann, which describes this exact scenario and appears to confirm our hypothesis.

Microsoft's fix involved scanning the PE signature block for known byte patterns that could indicate this type of abuse.

Investigating the false positive

Upon further debugging, we discovered that the binary was being flagged due to the signature data containing the `EGGA` marker from the list above:



EGGA marker

In the context of the list of markers above, the `EGGA` signature appears to relate to a specific header value used by an archive format called ALZip. Our code does not make any use of this file format.

Microsoft's heuristic treated the presence of `EGGA` as evidence that malicious archive data had been embedded in the PE signature. In practice, nothing of the sort was present. The signature block itself happened to include those four bytes as part of the hashed data.

Collisions like this are unusual, but page hashing (`/ph`) made it more likely. By expanding the size of the signature block, page hashing increases the surface area for coincidental matches and increases the likelihood of triggering the heuristic.

The binary didn't contain any self-extracting routines, so the hit on `EGGA` was a false positive. In that context, the warning had no bearing on the file's integrity. This meant it was safe to re-sign the file with `/rmc` to restore the expected validation.

Conclusion

It is well known that additional data can be embedded in a PE file without breaking its signature by appending it to the security block. Even some legitimate software products take advantage of this to embed user-specific metadata into signed executables. However, we were not aware that Microsoft had implemented heuristics to detect specific malicious cases of this, even though they were introduced back in 2012.

The original error message was very vague, and we were unable to find any documentation or references online that helped explain the behavior. Even searching for the associated registry value after discovering it (`PECertInvalidMarkers`) yielded zero results.

What we uncovered is that Microsoft added heuristic scanning of signature blocks more than a decade ago to counter specific abuse cases. Those heuristics reside in a hardcoded list of "invalid markers," many of which are tied to outdated installers and archive formats. Our binary happened to collide with one of those markers when signed with page hashing enabled, creating a validation failure with no clear documentation and no public references to the underlying registry key or detection logic.

The absence of online discussions regarding this failure mode, aside from a single unresolved Visual Studio Developer Community post from 2018, made the initial diagnosis difficult. By publishing this analysis, we want to provide a technical reference point for others who may encounter the same problem. In our case, resolving the issue required deep troubleshooting that few outside this space would normally need to exercise. For teams automating code signing, the key lesson is to integrate signature validation checks early and be aware that heuristic marker detection can lead to edge-case failures.

Additional references

The author can be found on X at @x86matthew.