SSD Advisory – Bitdefender Code Signing organizationName Buffer Overflow

blogs.securiteam.com/index.php/archives/3211

SSD / Maor Schwartz May 18, 2017

Vulnerability Summary

The following advisory describes a Buffer Overflow vulnerability found in Bitdefender Engine PE.

Bitdefender provides the Bitdefender “antimalware” engine for integration with other security vendors products. The engine is used in Bitdefender’s own products, for example in Bitdefender Internet Security 2017 and below. The antimalware engine is the core of the product, among other features providing the means to scan potentially malicious portable executables (PEs).

Credit

An independent security researcher, Pagefault, has reported this vulnerability to Beyond Security’s SecuriTeam Secure Disclosure program.

Vendor Response

Bitdefender has released patched to address this vulnerability in version 7.71417.

Vulnerability Details

A PE file can be signed using X.509 certificates. The certificates can ensure that the content of the executable has not been altered and that the executable comes from a trusted source.

Certificates are embedded inside one of the PE data directories defined via IMAGE_NT_HEADERS.IMAGE_OPTIONAL_HEADER.

The IMAGE_NT_HEADERS structure inside a PE file starts with the “PE\0\0” signature:

```
1  typedef struct _IMAGE_NT_HEADERS {
2      DWORD Signature; "PE\0\0"
3      IMAGE_FILE_HEADER FileHeader;
4      IMAGE_OPTIONAL_HEADER OptionalHeader;
5  } IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

The *IMAGE_OPTIONAL_HEADER* structure contains several *DataDirectory IMAGE_DATA_DIRECTORY* structures inside its last fields:

```
1    WORD Magic
2    BYTE MajorLinkerVersion
3    ...
4    DWORD LoaderFlags
5    DWORD NumberOfRvaAndSizes
6    IMAGE_DATA_DIRECTORY DataDirectory[16]
7    -------------------------------------------------
8    typedef struct _IMAGE_DATA_DIRECTORY {
9        DWORD   VirtualAddress;     // RVA of the data
10       DWORD   Size;               // Size of the data
11   };
```

DataDirectory[4] represents *IMAGE_DIRECTORY_ENTRY_SECURITY*, and points to a list of *WIN_CERTIFICATE* structures. The *VirtualAddress* field is a file offset, rather than an RVA.

The *WIN_CERTIFICATE* structures is defined as follows:

```
1  typedef struct _WIN_CERTIFICATE {
2    DWORD dwLength;
3    WORD  wRevision;
4    WORD  wCertificateType;
5    BYTE  bCertificate[ANYSIZE_ARRAY];
6  } WIN_CERTIFICATE, *PWIN_CERTIFICATE;
```

vsserv.exe is the Bitdefender system service. The process scans PEs automatically, analyzing digital signatures through the *cevakrnl.rv8* module. The module is located in a compressed form under "*%ProgramFiles%\Common Files\Bitdefender\Bitdefender Threat Scanner\Antivirus_…\Plugins*".

cevakrnl.rv8 is unpacked and loaded as executable code on service startup. *cevakrnl.rv8!sub_40ACFF0()* is called when a signed PE is encountered.

```
1   cevakrnl.rv8:040AE691            lea     eax, [ebp+var_2C]
2   cevakrnl.rv8:040AE694            push    eax             ; &(ebp-0x2C) - object placed on the stack
3   cevakrnl.rv8:040AE695            call    sub_40ACFF0     ; call here
4
5   cevakrnl.rv8!sub_40ACFF0() extracts the IMAGE_DIRECTORY_ENTRY_SECURITY offset and size fields.
6
7   cevakrnl.rv8:040ACFF0 sub_40ACFF0     proc near               ; CODE XREF: sub_40AE5C0+D5p
8   cevakrnl.rv8:040ACFF0
9   ...
10  cevakrnl.rv8:040AD007             mov     edi, [ebp+arg_0]
11  ...
12  cevakrnl.rv8:040AD025             mov     eax, [edi+4]    ; eax = IMAGE_NT_HEADERS
13  cevakrnl.rv8:040AD025                                     ; contains at
14  cevakrnl.rv8:040AD025                                     ; offset  0x0: DWORD Signature ("PE");
15  cevakrnl.rv8:040AD025                                     ; offset  0x4: IMAGE_FILE_HEADER FileHeader;
16  cevakrnl.rv8:040AD025                                     ; offset 0x18: IMAGE_OPTIONAL_HEADER32
17  OptionalHeader;
18  cevakrnl.rv8:040AD028             mov     [ebp+arg_0_bkup], edi
19  cevakrnl.rv8:040AD02E             mov     [ebp+numofcrcs], ecx
20  cevakrnl.rv8:040AD034             mov     [ebp+var_1F0], ecx
21  cevakrnl.rv8:040AD03A             mov     esi, [eax+9Ch]  ; attribute certificate size
22  cevakrnl.rv8:040AD03A                                     ; OptionalHeader.DataDirectory+0x24
23  cevakrnl.rv8:040AD03A                                     ; = IMAGE_DIRECTORY_ENTRY_SECURITY.Size
24  cevakrnl.rv8:040AD040             mov     edx, [eax+98h]  ; attribute certificate offset
25  cevakrnl.rv8:040AD040                                     ; OptionalHeader.DataDirectory+0x20
26  cevakrnl.rv8:040AD040                                     ; = IMAGE_DIRECTORY_ENTRY_SECURITY.Offset
    cevakrnl.rv8:040AD040                                     ; "Points to a list of WIN_CERTIFICATE structures, defined in
    WinTrust.H"
```

A maximum number of *0x2400* bytes is then read from the defined offset into a heap buffer.

```
cevakrnl.rv8:040AD092              cmp     esi, 2400h      ; maximum size
cevakrnl.rv8:040AD098              jbe     short @max
cevakrnl.rv8:040AD09A              mov     esi, 2400h
cevakrnl.rv8:040AD09F @max:                                ; CODE XREF: sub_40ACFF0+A8j
...
cevakrnl.rv8:040AD0C4              lea     eax, [ebp+var_1C4]
cevakrnl.rv8:040AD0CA              push    eax             ; int
cevakrnl.rv8:040AD0CB              push    esi             ; size
cevakrnl.rv8:040AD0CC
cevakrnl.rv8:040AD0CC loc_40AD0CC:                         ; CODE XREF: sub_40ACFF0+CEj
cevakrnl.rv8:040AD0CC              mov     ebx, [ebp+buf]
cevakrnl.rv8:040AD0D2              mov     edi, [ebp+arg_0_bkup]
cevakrnl.rv8:040AD0D8              push    ebx             ; buf
cevakrnl.rv8:040AD0D9              push    edx             ; offset
cevakrnl.rv8:040AD0DA              push    edi             ; int
cevakrnl.rv8:040AD0DB              call    readatoffset    ; read all structures
cevakrnl.rv8:040AD0DB                                      ;   typedef struct _WIN_CERTIFICATE {
cevakrnl.rv8:040AD0DB                                      ;     DWORD dwLength;
cevakrnl.rv8:040AD0DB                                      ;     WORD wRevision;
cevakrnl.rv8:040AD0DB                                      ;     WORD wCertificateType;
cevakrnl.rv8:040AD0DB                                      ;     BYTE bCertificate[ANYSIZE_ARRAY];
cevakrnl.rv8:040AD0DB                                      ;   } WIN_CERTIFICATE,*LPWIN_CERTIFICATE;
```

After additional irrelevant operations, Bitdefender starts searching for X.509 "*organizationName*" attributes in encountered data. The attributes are located by searching for the 0x0A045503 dword, which is the ASN.1 representation of the *organizationName* OID 2.5.4.10.

```
cevakrnl.rv8:040AD320 @startloop:                          ; CODE XREF: sub_40ACFF0+326j
cevakrnl.rv8:040AD320                                      ; sub_40ACFF0+728j
cevakrnl.rv8:040AD320              mov     ecx, [ebp+buf]
cevakrnl.rv8:040AD326              mov     eax, [ecx+esi]  ; current dword
cevakrnl.rv8:040AD329              lea     ebx, [ecx+esi]
cevakrnl.rv8:040AD32C              mov     [ebp+var_208], ebx
cevakrnl.rv8:040AD332              cmp     eax, 0A045503h  ; 55:04:0A = X.509 "id-at-organizationName"
attribute
cevakrnl.rv8:040AD337              jz      short @found
```

When an "*organizationName*" is found, its corresponding value string is passed in a call to a CRC32-computing function. The function returns the inverted (bitwise NOT) CRC32 sum of the string.

Please note that only printable ASCII (0x20-0x7E) characters are considered valid in "*organizationName*".

```
1   cevakrnl.rv8:040AD3B8 @found:                          ; CODE XREF: sub_40ACFF0+347j
2   cevakrnl.rv8:040AD3B8                                  ; sub_40ACFF0+357j
3   cevakrnl.rv8:040AD3B8             mov     bl, [ecx+esi+5] ; value string length
4   cevakrnl.rv8:040AD3BC             movzx   eax, bl
5   cevakrnl.rv8:040AD3BF             mov     [ebp+var_20C], eax
6   cevakrnl.rv8:040AD3C5             add     eax, 6
7   cevakrnl.rv8:040AD3C8             add     eax, esi
8   cevakrnl.rv8:040AD3CA             mov     [ebp+var_1E8], 0
9   cevakrnl.rv8:040AD3D4             mov     [ebp+var_40], 0
10  cevakrnl.rv8:040AD3D8             mov     [ebp+savedcrc], 0
11  cevakrnl.rv8:040AD3E2             mov     [ebp+after_value_string], eax ; offset to next data
12  ...
13  cevakrnl.rv8:040AD444             mov     eax, [ebp+buf]
14  cevakrnl.rv8:040AD44A             add     eax, 6
15  cevakrnl.rv8:040AD44D             mov     [ebp+edi+var_40], 0
16  cevakrnl.rv8:040AD452             add     eax, esi        ; offset + 6
17  cevakrnl.rv8:040AD452                                    ; points to value string
18  cevakrnl.rv8:040AD454             push    edi             ; length of string
19  cevakrnl.rv8:040AD455             push    eax             ; Organization in certificate
20  cevakrnl.rv8:040AD456             call    crc32           ; crc32
21  cevakrnl.rv8:040AD45B             add     esp, 8          ; this returns ~crc32
22  cevakrnl.rv8:040AD45B                                    ; ~crc32("31TZnp") = 0xdeadbeef
```

If the CRC was not previously encountered:

```
1   cevakrnl.rv8:040AD480 @checkduplicate:                 ; CODE XREF: sub_40ACFF0+488j
2   cevakrnl.rv8:040AD480                                  ; sub_40ACFF0+4A0j
3   cevakrnl.rv8:040AD480             cmp     [ebp+ecx*4+crc32results], eax ; array of already saved CRCs
4   cevakrnl.rv8:040AD487             jz      @duplicate
5   cevakrnl.rv8:040AD48D             inc     ecx
6   cevakrnl.rv8:040AD48E             cmp     ecx, ebx
7   cevakrnl.rv8:040AD490             jb      short @checkduplicate
```

its value is placed inside a local stack array of 8 dwords. The index of the array is increased for each unique CRC without checking the array limit. This results in a stack-based buffer overflow if an overly large number of unique "*organizationName*" values are encountered.

```
1   -000001B8 crc32results    dd 8 dup(?)
2   -00000198 var_198         db 256 dup(?)
3   ...
4   cevakrnl.rv8:040AD51E             mov     eax, [ebp+savedcrc]
5   cevakrnl.rv8:040AD524             mov     [ebp+ebx*4+crc32results], eax ; buffer overflow
6   cevakrnl.rv8:040AD524                                    ; [ebp+ebx*4-0x1B8] = eax
7   cevakrnl.rv8:040AD52B             inc     ebx
8   cevakrnl.rv8:040AD52C             mov     [ebp+numofcrcs], ebx
```

The vulnerability allows overwriting a large number of stack bytes with arbitrary data. The data written to the stack is arbitrary due to the ability to find an ASCII string for any desired CRC result by reversing the CRC32 algorithm.

Although the vulnerable function contains a cookie check on return, code execution is believed possible due to the use of an object placed on the stack prior to function return.

The object is passed to the vulnerable function as the first argument, and the field at offset 0x1C (changed to 0xdeadbeef via the PoC) is passed to *global_function0()*.

```
1  cevakrnl.rv8:040AD750          mov     ebx, [ebp+arg_0_bkup] ; ebx points to the stack of the caller function,
2  ; which is above crc32results
3  ...
4  cevakrnl.rv8:040AD785          push    0
5  cevakrnl.rv8:040AD787          push    1
6  cevakrnl.rv8:040AD789          push    41C40Eh
7  cevakrnl.rv8:040AD78E          push    6
8  cevakrnl.rv8:040AD790          push    dword ptr [ebx+1Ch] ; corrupted
9  cevakrnl.rv8:040AD793          call    global_function0
```

global_function0() calls *sub_2F70B90()*, passing [0xdeadbeef+0x22C] as the current object.

```
1  seg001:02F5D69F          mov     ecx, [ecx+22Ch] ; crash here
2  seg001:02F5D69F                                  ; ecx is controlled
3  seg001:02F5D6A5          push    [ebp+arg_4]
4  seg001:02F5D6A8          call    sub_2F70B90
```

sub_2F70B90() extracts a dword from the current object pointer

```
1  seg001:02F70BFA          mov     edi, [esi+eax*4] ; eax - fixed offset = 0x560
```

eventually passing it as the current object to *sub_2F6F120()*

```
1  seg001:02F70D45          mov     ecx, edi
2  seg001:02F70D47          call    sub_2F6F120
```

sub_2F6F120() eventually extracts a dword from the potentially arbitrary pointer, resulting in a jump to an arbitrary address.

```
1  seg001:02F6F132          mov     eax, [edi+4]
2  seg001:02F6F135          push    ebx
3  seg001:02F6F136          push    dword ptr [esi+4]
4  seg001:02F6F139          push    edi
5  seg001:02F6F13A          call    eax
```

The ability to jump to an arbitrary address depends on the ability to place controlled content at a fixed address. Heap spraying could be used for this purpose. This is believed achievable due to the complexity of the Bitdefender engine.