Wired for Management Review Request bis037
Identify required signer in BIS_GetSignatureInfo

CLOSED	1	02-01-1999	8-4-1999
Status	**Priority**	**Received**	**Complete**
BIS	BIS V1.0	BIS V1.0	1.0 errata
Specification	**Release**	**Target Specification**	**Target Release**
Paul Drews	Intel	(503) 264-8488	paul.drews@intel.com
Requester	**Company**	**Voice**	**Email**

The following text describes an approved change to the Boot Integrity Services Application Programming Interface Version 1.0 Dated December 28, 1998. Please substitute the Changed Text given below in the appropriate section of the specification. If you are interested in the rationale and history behind this Review Request, this information follows at the end of the document.

Changed Text:
Replace the entire text of section 2.3 "Remote-boot authentication" with the following text:

2.3 Remote-boot authentication

A typical Remote-boot authentication example begins with activities outside the platform boot sequence. The IT organization that manages the platform(s) configures the Boot Object Authorization Certificate for the platform as described previously.

The IT organization evaluates a boot image and decides that it should be authorized to execute on the managed platform(s). The IT organization generates several boot image credential files for the boot image, one for each of the signature algorithm and key-length combinations mandated by this specification. The boot image credential is a signed manifest. The format of this credential and the tools to generate one are described in more detail in the associated Software Development Kit (SDK) specification. For the purpose of this example, the manifest includes the following:

- Integrity data for the boot image.
- The signing IT organization's certificate. This is the IT organization's Boot Object Authorization Certificate.
- A digital signature of the entire manifest. This digital signature is generated using the signing organization's (secret) private key. This private key corresponds to the public key found in the signer's certificate.

The IT organization puts the boot image file and the corresponding boot image credential files on a Boot Server accessible to the managed platform(s). The managed platform must know how to locate the boot image credential files corresponding to a boot image on the Boot Server.

Now that the boot image and corresponding boot image credential files are stored on a Boot Server, the example shifts to the managed platform boot sequence. A typical boot sequence starts becoming interesting to Remote-Boot authentication late in the BIOS initialization sequence. At a high level, the relevant sequence of typical steps are:

1. The remote-boot code for a Network Interface Card uses Dynamic Host Configuration Protocol (DHCP) to obtain a platform IP address, a Boot Server IP address, and a boot file name.
2. The remote-boot code uses the Preboot Execution Environment (PXE) code to download a boot file.
3. The remote-boot code uses the BIS_GetSignatureInfo function described in this document to find a digital certificate, signature algorithm, and key-length combination that the platform

supports. This also indirectly determines which of the corresponding boot image credential files to download from the Boot Server.

4. The remote-boot code downloads the corresponding boot image credential.
5. The remote-boot code calls the BIS_VerifyBootObject function described in this document to perform the integrity and authorization check of the image. The integrity check must succeed as described in detail in the function specification. The authorization check involves checking the signer's certificate supplied in the credential. The public key in the certificate is compared against the public key in the Boot Object Authorization Certificate configured for this platform. If a match is found, the signature was generated by the accepted authority and the authorization check passes, otherwise the boot attempt fails.
6. Assuming all checks in the previous step pass, the remote-boot code branches to the downloaded boot image, which never returns.
7. This first downloaded boot image is subject to fairly tight size and memory-model constraints. Consequently in the typical example it is merely a first-stage bootstrap. It contains a rudimentary memory manager to make additional memory space available, and a more robust download protocol that can take advantage of the expanded memory. It downloads a second-stage bootstrap using a server, protocol, and file location that may be determined from information obtained in the first-stage download.
8. The second-stage boot image has its own integrity and authorization credentials, separate from the first-stage boot image. The first stage code may use the BIS_VerifyObjectWithCredential function to validate the second stage. This function is similar to the BIS_VerifyBootObject function except that it allows the caller to supply a certificate that shall be recognized as the source of authority. In particular, the source of authority for second-stage signature is typically the vendor of the software being booted, not the IT organization that manages the platform.
9. Assuming the checks in the previous step pass, the first stage code invokes the second stage code, which never returns.
10. The bootstrap process may involve several more stages, with each stage downloading the credential and image of the next, and validating it before executing it.
11. Eventually, an OS typically has enough capabilities initialized that it no longer needs the BIS service and can take advantage of the memory space that the BIS service occupies. To ensure clean termination of the BIS service, the client code calls a BIS_Shutdown function to terminate the service. The memory occupied by the BIS service may then be reclaimed, managed, and over-written by an OS. This makes the BIS service no longer available.

Most of this usage model example involves activities outside the direct scope of the interfaces defined or referenced in this document. The parts actually within this interface scope are:

- The Boot Object Authorization Certificate and Boot Authorization Check Flag as platform configuration parameters.
- The generalized BIS_UpdateBootObjectAuthorization function for storing these parameters in a persistent and protected manner.
- A boot image credential including the certificate used to sign the boot image credentials. The public key in the signer's certificate must match the public key in the Boot Object Authorization Certificate of this platform.
- BIS_GetSignatureInfo function to find a digital certificate, signature algorithm, and key-length combination that the platform supports.
- BIS_VerifyBootObject function to test integrity and authorization of the boot image.

- BIS_VerifyObjectWithCredential function for implementing a more customized authentication model.
- BIS_Shutdown function for terminating BIS usage and preparing to reclaim its resources.

It is the responsibility of other parts of software inside and outside of the managed platform to perform the remaining parts of the usage model.

Changed Text:
Replace the entire text of section 3.8.5 “BIS_ALG_ID” with the following text:

3.8.5 BIS_ALG_ID

```
typedef UINT16                    BIS_ALG_ID;
```

This type represents a digital signature algorithm. A digital signature algorithm is often composed of a particular combination of secure hash algorithm and encryption algorithm. This type also allows for digital signature algorithms that cannot be decomposed.

Changed Text:
Insert the Sections 3.8.16 “BIS_CERT_ID”, 3.8.17 “BIS_CERT_ID_MASK”, and 3.8.18 “Predefined BIS_CERT_ID values” before Section 3.8.16 “BIS_SIGNATURE_INFO”. Renumber the old 3.8.16 and following sections appropriately. The text of the sections to be inserted is as follows:

3.8.16 BIS_CERT_ID

```
typedef UINT32   BIS_CERT_ID;
```

This type represents a shortened value that identifies the platform’s currently configured Boot Object Authorization Certificate. The value is the first four bytes, in “little-endian” order, of the SHA-1 hash of the certificate, except that the most-significant bits of the second and third bytes are reserved, and must be set to zero regardless of the outcome of the hash function. This type is included in the array of values returned from the BIS_GetSignatureInfo function to indicate the required source of a signature for a boot object or a configuration update request. There are a few predefined reserved values with special meanings as described below.

3.8.17 BIS_CERT_ID_MASK

```
#define BIS_CERT_ID_MASK (0xFF7F7FFF)
```

This C preprocessor symbol may be used as a bit-wise “AND” value to transform the first four bytes (in little-endian order) of a SHA-1 hash of a certificate into a certificate ID with the “reserved” bits properly set to zero.

3.8.18 Predefined BIS_CERT_ID values

```
#define BIS_CERT_ID_DSA      BIS_ALG_DSA     //CSSM_ALGID_DSA
#define BIS_CERT_ID_RSA_MD5  BIS_ALG_RSA_MD5 //CSSM_ALGID_MD5_WITH_RSA
```

These C preprocessor symbols provide values for the BIS_CERT_ID type. These values are used when the platform has no configured Boot Object Authorization Certificate. They indicate the signature algorithm and key length that is supported by the platform. Users must be careful to avoid constructing Boot Object Authorization Certificates that transform to BIS_CERT_ID values that collide with these predefined values or with the BIS_CERT_ID values of other Boot Object Authorization Certificates they use.

Changed Text:

Replace the entire Section 3.8.16 “BIS_SIGNATURE_INFO” with the following text (renumbered to Section 3.8.19):

3.8.19 BIS_SIGNATURE_INFO

```
typedef struct _BIS_SIGNATURE_INFO
{

    BIS_CERT_ID   certificateID;  // A truncated hash of the
                                  // platform’s Boot Object
                                  // Authorization Certificate.
    BIS_ALG_ID    algorithmID;    //A signature algorithm number.
    UINT16        keyLength;      //Length of alg. keys in bits.
}
BIS_SIGNATURE_INFO;

#if defined(COMPILER_IS_32_BIT)
typedef struct _BIS_SIGNATURE_INFO *BIS_SIGNATURE_INFO_PTR;
#endif

#if defined(COMPILER_IS_16_BIT)
typedef UINT32 BIS_SIGNATURE_INFO_PTR;
#endif
```

This type defines a digital certificate, digital signature algorithm, and key-length combination that may be supported by the BIS implementation. This type is returned by BIS_GetSignatureInfo to describe the combination(s) supported by the implementation.

Definitions:

certificateID - A shortened value identifying the platform’s currently configured Boot Object Authorization Certificate, if one is currently configured. The shortened value is derived from the certificate as defined in section 3.8.16, “BIS_CERT_ID”. If there is no certificate currently configured, the value is one of the reserved BIS_CERT_ID_XXX values defined above.

algorithmID - A predefined constant representing a particular digital signature algorithm. Often this represents a combination of hash algorithm and encryption algorithm, however, it may also represent a standalone digital signature algorithm.

keyLength - The length of the public key, in bits, supported by this digital signature algorithm.

The 16-bit pointer type is declared as a 32-bit unsigned integer instead of a pointer because the actual value used by the BIS interface is a 32-bit physical address. Any necessary translation between physical address and 16-bit segment:offset addressing must be accomplished by the caller.

Changed Text:
Replace entire Section 3.8.17 “BIS_GET_SIGINFO_COUNT” with the following text (renumbered to Section 3.8.20):

3.8.20 BIS_GET_SIGINFO_COUNT

```
#define BIS_GET_SIGINFO_COUNT(bisDataPtr)                  \
        ((bisDataPtr)->length/sizeof(BIS_SIGNATURE_INFO))
```

This macro computes how many BIS_SIGNATURE_INFO elements are contained in a BIS_DATA structure returned from BIS_GetSignatureInfo. It represents the list of supported digital certificate, digital signature algorithm, and key-length combinations.

Definitions:

bisDataPtr - Supplies the 32-bit physical-address pointer of the target BIS_DATA structure.

(return value) - The number of BIS_SIGNATURE_INFO elements contained in the array.

This macro is supplied for use only in the 32-bit compilation environment. In the 16-bit compilation environment, it is the caller's responsibility to accomplish this computation, including any necessary translation between physical address and 16-bit segment:offset addressing.

Changed Text:
Replace the entire Section 3.18.18 "BIS_GET_SIGINFO_ARRAY" with the following text (renumbered to Section 3.18.21):

3.8.21 BIS_GET_SIGINFO_ARRAY

```
#define BIS_GET_SIGINFO_ARRAY(bisDataPtr)                \
        ((BIS_SIGNATURE_INFO_PTR)(bisDataPtr)->data)
```

This macro returns a 32-bit physical address pointer to the BIS_SIGNATURE_INFO array contained in a BIS_DATA structure returned from BIS_GetSignatureInfo representing the list of supported digital certificate, digital signature algorithm, and key-length combinations.

Definitions:

bisDataPtr - Supplies the 32-bit physical-address pointer of the target BIS_DATA structure.

(return value) - The 32-bit physical address pointer to the BIS_SIGNATURE_INFO array, cast as a BIS_SIGNATURE_INFO_PTR.

This macro is supplied for use only in the 32-bit compilation environment. In the 16-bit compilation environment, it is the caller's responsibility to accomplish this computation, including any necessary translation between physical address and 16-bit segment:offset addressing.

Changed Text:
Replace the entire Section 3.11 "Initialization, Shutdown, and Utility functions" up to but not including Section 3.11.1 "BIS_Initialize" with the following text:

3.11 Initialization, Shutdown, and Utility functions

This group of functions consists of miscellaneous support functions used for initialization, shutdown, information, and freeing memory allocated and returned by other functions. The functions in this group are:

BIS_Initialize - Initializes the BIS service, checking that it is compatible with the version requested by the caller. After this call, other BIS functions may be invoked.

BIS_Shutdown - Shuts down the BIS service. After this call, other BIS functions may no longer be invoked.

BIS_Free - Deallocates a BIS_DATA_PTR and associated memory allocated by one of the other BIS functions.

BIS_GetSignatureInfo - Retrieves a list of digital certificate, digital signature algorithm, hash algorithm, and key-lengths that the platform supports.

The functions in this group are described in detail in the next sections.

Changed Text:
Replace the entire Section 3.11.4 "BIS_GetSignatureInfo" with the following text:

3.11.4 BIS_GetSignatureInfo

```
typedef
struct _BIS_GetSignatureInfo_PARMS
{
    UINT32                 sizeOfStruct;   //[in]  Byte length of this
                                           //structure.
    BIS_STATUS             returnValue;    //[out] BIS_OK | error code.
    BIS_APPLICATION_HANDLE appHandle;      //[in]  From BIS_Initialize( ).
```

```
    BIS_DATA_PTR            signatureInfo;  //[out] Signature info struct.
}
BIS_GSI_PARMS;
```

Calling Example:

```
UINT8                     integrityOk;
BIS_ENTRY_POINT           entryStructure;
BIS_GSI_PARMS             params;
BIS_APPLICATION_HANDLE    appHandle;
. . .
// Set "in" parameter values
params.sizeOfStruct = sizeof(params);
params.appHandle    = appHandle;
// Invoke operation through entry point procedure pointer
integrityOk = (* entryStructure.bisEntry32)(
    BISOP_GetSignatureInfo,   // opCode
    (void *) & params,        // pParamBundle
    BIS_TRUE);                // checkFlag
```

The function selected with the BISOP_GetSignatureInfo operation code retrieves a list of digital certificate identifier, digital signature algorithm, hash algorithm, and key-length combinations that the platform supports. The list is an array of (certificate id, algorithm id, key length) triples, where the certificate id is derived from the platform's Boot Object Authorization Certificate as described in section 3.8.16, "BIS_CERT_ID", the algorithm id represents the combination of signature algorithm and hash algorithm, and the key length is expressed in bits. The number of array elements can be computed using the Length field of the retrieved BIS_DATA_PTR.

The retrieved list is in order of preference. A digital signature algorithm for which the platform has a currently configured Boot Object Authorization Certificate is preferred over any digital signature algorithm for which there is not a currently configured Boot Object Authorization Certificate. Thus the first element in the list has a certificateID representing a Boot Object Authorization Certificate if the platform has one configured. Otherwise the certificateID of the first element in the list is one of the reserved values representing a digital signature algorithm.

bisEntry32 or bisEntry16 Parameters

opCode (input) - The manifest constant BISOP_GetSignatureInfo.

pParamBundle (input, output) - A 32-bit physical address pointer to a specific parameter bundle structure of type BIS_GSI_PARMS as described below.

checkFlag (input) - A boolean value indicating whether or not an internal integrity check should be performed before performing the specific operation. If this value is non-zero (BIS_TRUE) the check should be performed.

bisEntry32 or bisEntry16 Return Value

Zero - No integrity fault was detected.

Non-zero - An integrity fault of some sort was detected and the operation was not performed. "Out" field values in the parameter bundle structure are not valid.

Parameter Bundle Fields

sizeOfStruct (input) - The length of the BIS_GSI_PARMS structure, in bytes.

returnValue (output) - If the specific operation is successful, the function writes the value BIS_OK. Otherwise, the function writes a non-zero error code indicating the detailed error that occurred.

appHandle (input) - An opaque handle that identifies the caller's instance of initialization of the BIS service.

signatureInfo (output) - The function writes an allocated BIS_DATA_PTR containing the array of BIS_SIGNATURE_INFO structures representing the supported digital certificate identifier, algorithm, and key length combinations. The caller must eventually free the memory allocated by this function using the function BIS_Free.

See Also

BIS_Free

Review Request History -*this section documents the discussion behind this Review Request. It shows the rationale behind the proposal, including the reason the fix was desired and why the particular fix was chosen.*

****Only those changes explicitly in the **Changed Text** section above have been approved and are now officially a part of the specification.****

Summary - When PXE and BIS are integrated, PXE puts (a subset of) the compact information from BIS_GetSignatureInfo into a DHCP packet that requests the Boot Image signature file name from the boot server. However, the BIS_GetSignatureInfo information identifies only the required signature algorithm, not the required signer identity out of possibly multiple signers that use this signature algorithm. This proposal would add data into the return from BIS_GetSignatureInfo to identify the required signer.

Background This problem was encountered while working through an integration scenario between PXE and BIS. The typical usage scenario described for BIS involves small groups of platforms, where each group of platforms is managed by a distinct Information Technology authority, represented by a distinct cryptographic key. Each of the platforms within a group gets configured with the Boot Object Authorization Certificate containing the public key of this authority that the platform shall recognize as the source of authority for Boot Object signatures.

Meanwhile, the PXE boot algorithm determines whether BIS is present and enabled on the platform. If present and enabled, the PXE boot algorithm calls BIS_GetSignatureInfo to retrieve information about the digital signature capabilities supported by BIS in the platform. The returned information is an array, where each array element is a pair of 4-byte items that identify (1) the digital signature algorithm, and (2) the key-length to be used within that algorithm. PXE ignores the key-length parts of the pairs and packs the 4-byte signature algorithm identifiers into a DHCP request sent to the PXE-enabled boot server.

For every boot image that the PXE-enabled boot server can supply, the boot server also has a list of digital signature files it can supply. The PXE-enabled boot server selects the digital signature filename it puts in the DHCP response based on a 4-byte signature algorithm identifier retrieved from the DHCP request.

The mismatch between the BIS scenario and the PXE scenario becomes evident when one considers a medium to large-sized installation. In such an installation, it is likely that platform management responsibility is partitioned into small domains of, say, a few dozen platforms managed by a each different Boot Object Authorization Certificate. At the same time, it is likely that a single PXE-enabled boot server will be called on to serve the needs of hundreds of platforms. Several different platforms using the same digital signature algorithm but with different Boot Object Authorization Certificates installed, may all send their DHCP requests to the same PXE-enabled boot server. The PXE-enabled boot server has no information in the DHCP request that directly distinguishes which of several boot image signature files, generated by different authorities, is the appropriate one to satisfy the DHCP request.

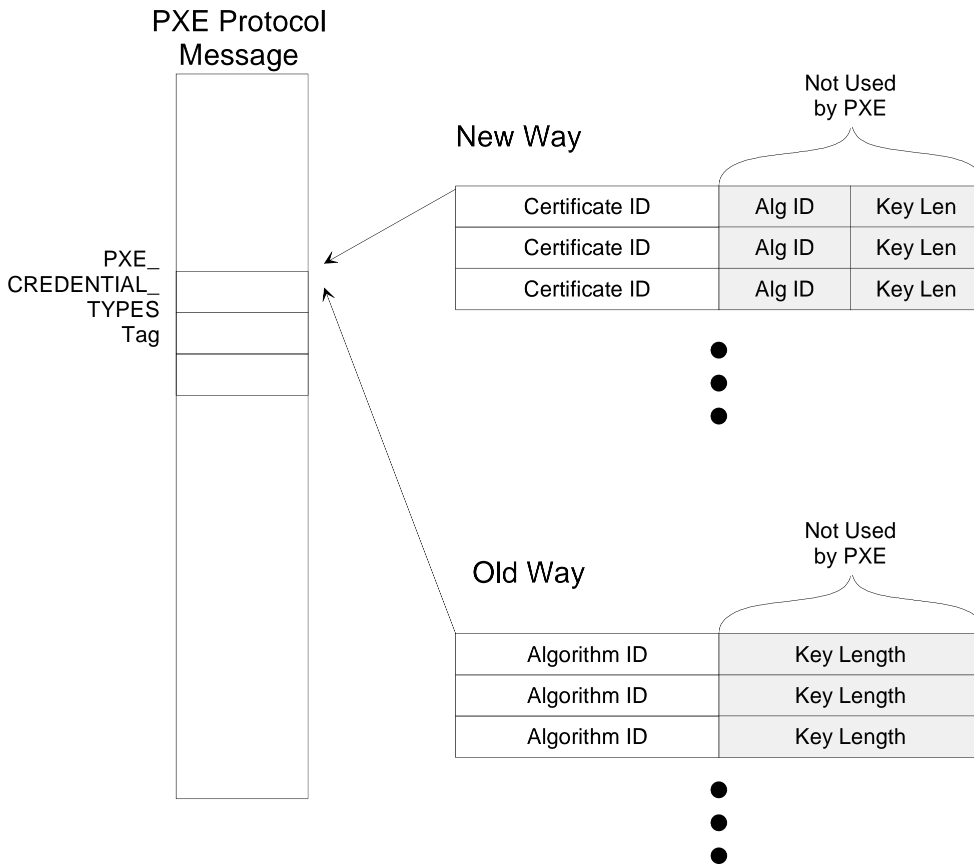
There are several different ways this problem could be solved. However, several considerations led to this particular choice of solutions to propose:

(1) Any mechanism that requires passing and maintaining platform/signer relationship information outside of the standard PXE and BIS mechanisms carries the risk of getting out of sync with respect to the information passed and maintained within PXE and BIS.

(2) The PXE client firmware was expecting first production release within a few days at the time the problem was identified. Any change to PXE client firmware at this late date should be avoided.

This proposal takes advantage of the existing PXE client firmware by re-organizing the structures returned from BIS_GetSignatureInfo. PXE uses only the first 4 bytes of each 8-byte array element, packing these into the PXE_CREDENTIAL_TYPES tag of a DHCPREQUEST packet. So this proposal squeezes the old

information into the last 4 bytes, then re-defines the first 4 bytes to be a "certificate identifier" that is carried through the PXE protocol by the existing PXE client firmware instead. Since this part of the information carried through the PXE protocol was very loosely defined, this still fits well within the definition of the contents of the PXE protocol. The newly-defined "certificate identifier" is a 4-byte hash of the desired signing certificate for this platform as defined in detail in the attached Review Request change text. Using a hash value here keeps the identifier to a short fixed length while still providing a high probability of providing unique certificate identifiers. A graphical view of the old and new structures and usage is shown below:



Amendment - As of 03-18-1999, another issue was encountered. The PXE client software build 67 encounters a failure if either of two particular bit positions have a "1" value in them. The problem was traced to a code-generation bug and a fix put in place for subsequent PXE client releases. Nevertheless it is possible that there are products in the field derived from PXE client software build 67, so this Review Request was amended to set these particular bits to "reserved, must be zero". This amendment was done before the Review Request was submitted to the reviewers.

Status - This section summarizes the milestones in the proposal and approval of this review request.

02-01-1999 Original discovery of problem and proposed solution internally to PXE engineers.

03-18-1999 Another BIS/PXE integration problem discovered and amendment added to proposal.

03-18-1999 Review Request published to Wired for Management review list for public comment.

07-29-1999 Final deadline set at 08-04-1999 for public comment and approval published to WfM review list.

08-04-1999 Final approval (no comments received by deadline) Review Request is CLOSED and fully approved as a corrigendum to the Boot Integrity Services Application Programming Interface Version 1.0 specification.