Bootloader Signed Block Stream of Commands

(BSBSC, version 1.0)

Reto Buerki, codelabs GmbH

January 2020

Contents

1 Introduction **2**

2 Command Stream Loader (CSL) **3**
2.1 Write Data to Address . . . 4
2.1.1 Purpose . . . 4
2.1.2 Structure . . . 4
2.2 Fill Region with Pattern . . . 4
2.2.1 Purpose . . . 4
2.2.2 Structure . . . 5
2.3 Set Entry Point . . . 5
2.3.1 Purpose . . . 5
2.3.2 Structure . . . 5
2.4 CPUID Checks . . . 5
2.4.1 Purpose . . . 5
2.4.2 Structure . . . 6

3 Signed Block Stream (SBS) **6**
3.1 Image Creation . . . 7
3.2 Image Verification . . . 7
3.3 Header . . . 8
3.4 Signature . . . 10
3.5 Blocks . . . 10
3.6 Hash Concatenation . . . 10
3.7 Padding . . . 10

4 Reference Implementation **11**

1 Introduction

The protocol specified in this document addresses the following issues when loading OS kernels, or more generally operating system images, via bootloaders:

1. Efficient integrity protection and data origin authentication
2. Checking of hardware/system state before boot
3. Loading data to arbitrary/specific memory locations

The first item is often referred to as secure boot. The overall goal is to only boot unaltered images from trusted sources. This is commonly achieved via cryptographic schemes like digital signatures and support from the hardware (e.g. by providing read-only storage for cryptographic keys).

Before booting a system on a target machine, it makes sense to check whether its hardware meets the assumptions of the system to boot. It is beneficial to do this at an early stage of the boot process, to minimize code execution in case a precondition is not met.

The 3rd item addresses the problem of *memory holes* in the physical address space of a target machine when loading large system images. RAM is often not consecutive because the BIOS may reserve certain regions for ACPI tables, NVRAM storage etc. In order to place a large image,

which spans reserved regions, a mechanism must exist to instruct the bootloader to load data chunks of given size to a specified memory address.

The protocol outlined in this document consists of two sub-protocols, which can be used independently or stacked.

In order to instruct the bootloader to perform certain actions in a modular and extensible way, a *Command Stream Loader (CSL)* protocol for the x86 architecture is introduced. This is explained in the following section 2.

To assert the origin and guarantee the integrity of a batch of commands, the signature of the data must be checked efficiently before executing the encoded commands. This functionality is provided by the *Signed Block Stream (SBS)* protocol specified in section 3.

2 Command Stream Loader (CSL)

A command stream is a Type-Length-Value (TLV) encoded stream of commands which are intended to be executed in-order by a command processing module in the bootloader. Figure 1 shows an example command stream with four commands and the internal structure of one command in detail.

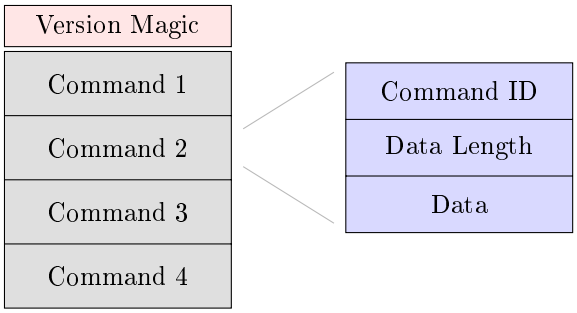


Figure 1: Stream of TLV-encoded commands

The *Command ID* field is 16-bits wide and encodes the command to be executed by the bootloader. The current protocol version defines the commands listed in table 1.

ID	Constant	Description
0	CMD_WRITE	Write data to physical address
1	CMD_FILL	Fill memory region with pattern
2	CMD_SET_ENTRY_POINT	Set kernel/image entry point
3	CMD_CHECK_CPUID	Perform `CPUID`-based system check

Table 1: CSL command IDs

With this mechanism in place, the bootloader command processing unit is able to perform any action defined in the CSL specification, making it a flexible and extensible mechanism to load system images for boot.

CSL images are identified by the version magic `0x8adc5fa2448cb65e` in the first eight bytes, little endian. A CSL implementation is required to check whether this sequence is present before processing commands.

The command ID range `60000 .. 65535` is reserved for vendor specific commands, which are not part of the official CSL specification. A CSL implementation may ignore commands in this range. It is the responsibility of the person choosing the custom command ID to assure its uniqueness in the respective context, in spite of the absence of any central registry for IDs.

A CSL implementation is required to check whether the specified data length in the header matches the expected length of the given command. Certain commands have a fixed data length, others have a minimum length.

Command parameters, e.g. the physical address of the `CMD_WRITE` command, are part of the data section. The data length field in the header specifies the sum of the lengths of the actual data and all parameters of a command.

Strings delivered as TLV data (e.g. check strings in the `CMD_CHECK_CPUID` command) must be Null-terminated. See the following sections for details about the structure and purpose of each command.

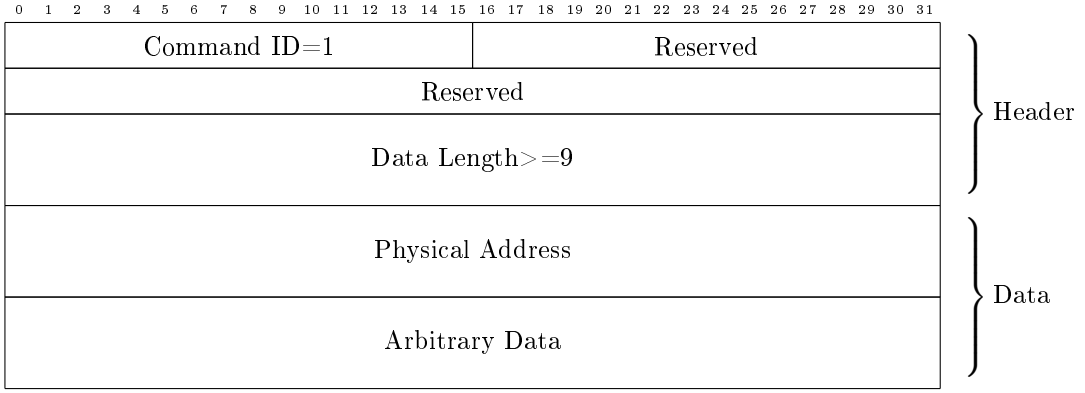
2.1 Write Data to Address

2.1.1 Purpose

The `CMD_WRITE` command instructs the bootloader to copy the data part of the command to the physical address specified by the address parameter. The amount of data to write to memory is the data length stored in the command header minus the size of the physical address parameter (which is eight bytes).

A CSL implementation is required to check whether the specified address can be reached in the active addressing mode, and raise an error and abort if not. It also needs to check whether the address lies within addressable RAM.

2.1.2 Structure



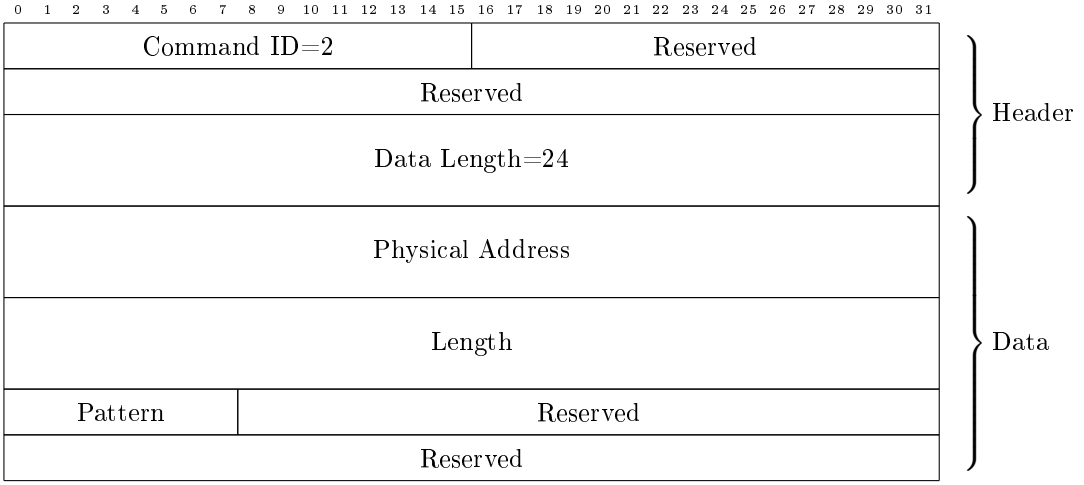
2.2 Fill Region with Pattern

2.2.1 Purpose

The `CMD_FILL` command consists of the physical address, length and pattern arguments. It directs the bootloader implementing the CSL protocol to fill the physical memory region spanned by address and length parameters with the specified byte pattern.

A CSL implementation is required to check whether the specified address can be reached in the active addressing mode, and raise an error and abort if not. Furthermore, it needs to check whether the memory region is within addressable RAM.

2.2.2 Structure



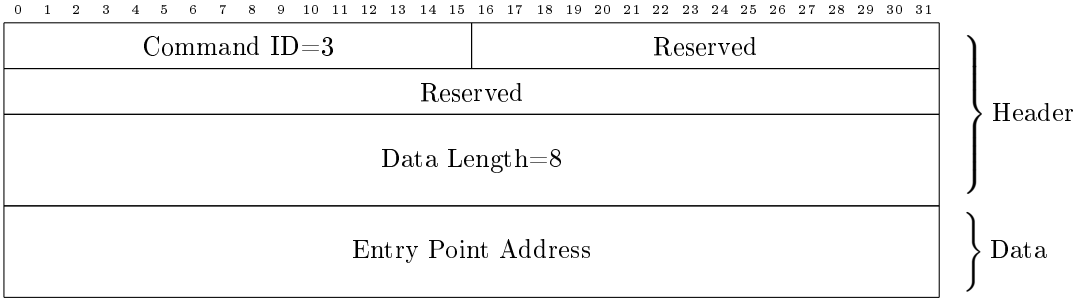
2.3 Set Entry Point

2.3.1 Purpose

After the OS kernel or system image has been constructed in memory via the `CMD_WRITE` and `CMD_FILL` commands, the bootloader must know the address to handoff execution, i.e. the entry point into the OS code. This information is delivered to the CSL implementation via the `SET_ENTRY_POINT` command. A bootloader sets the `Entry Point Address` value of this command as the `eip` or `rip` value in its boot state. All other registers of the state structure must be set to zero.

A CSL implementation is required to check whether the specified address can be reached in the active addressing mode, and raise an error and abort if not.

2.3.2 Structure



2.4 CPUID Checks

2.4.1 Purpose

`CPUID`-based checks are a way to make sure the system image encoded in the CSL format is deployed on a hardware platform supporting the required CPU features.

The `CPUID` instruction returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases also

ECX). A CSL implementation must check whether the `CPUID` instruction is available on the target hardware, if `CPUID` commands are encoded in the CSL image. The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction.

The `EAX Value` and `ECX Value` parameters specify the input to the `CPUID` operation. After execution, the `Mask` is applied to the register specified by the `Result Register` field (table 2) and compared to the given `Value`. If the value is equal, the check is passed. If not, the implementation must raise an error and abort.

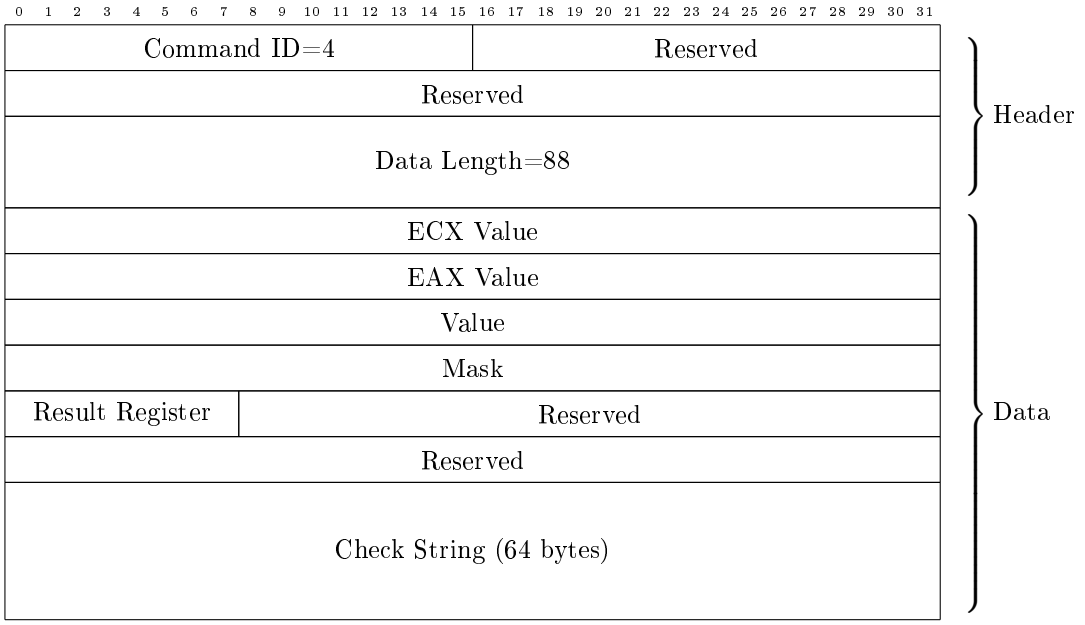
The 64-bytes check string is used to indicate to the user what check is performed in human readable form. This string must be Null-terminated. A CSL implementation must either check whether the string is Null-terminated and abort if not, or it may enforce it by allocating a character array of 64 bytes and explicitly setting the last byte to zero.

Future CSL protocol revisions may provide additional commands to check arbitrary memory locations, MSRs, control registers and the EFER register.

ID	Register
0	EAX
1	EBX
2	ECX
3	EDX

Table 2: CPUID result register encoding

2.4.2 Structure



3 Signed Block Stream (SBS)

The Signed Block Stream protocol described in this section is a mechanism to verify the origin and integrity of a command stream image described in the previous section. This is particu-

larly important because the CSL implementation executes encoded commands. However, such a mechanism is generally important in a secure boot context.

Note that the concept described in this section does not depend on the CSL protocol, it can be used to integrity protect any type of data.

3.1 Image Creation

Figure 2 outlines the principal operation of SBS. An input file is split into equal chunks of data of a given size. A block of `Block Size` encompasses the data block from the file plus a hashsum. The `Hash` fields in the blocks specify the `Hash of the Next Block`.

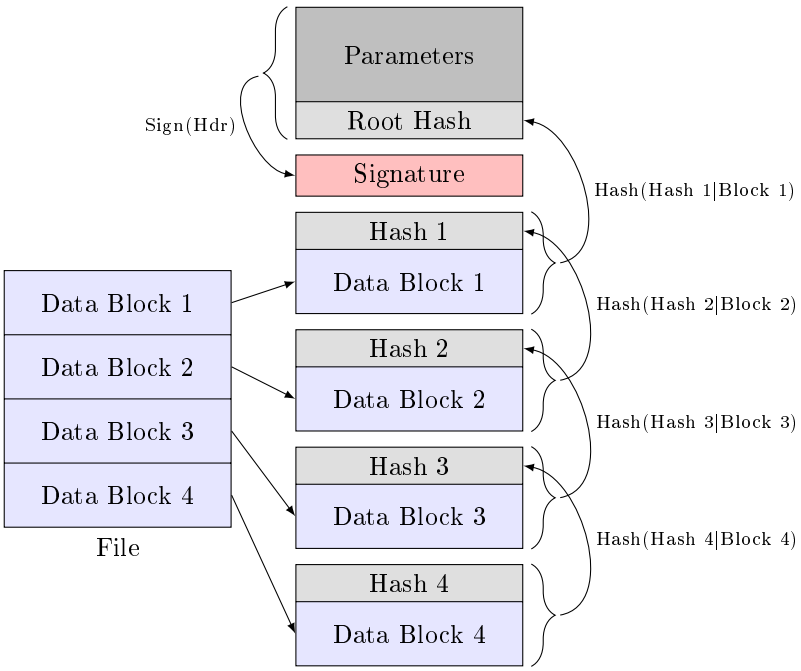


Figure 2: Hashing and signing operations

The hash of the last block must be zeroed, as there is no next block. The file data is read in reverse order and packed into blocks, whereas each block contains the hashsum over the next block data and hashsum value, as illustrated in figure 2. Reading the file from the end to the start allows to directly calculate the hashsums on the go, no second pass is required.

The block stream is prepended by signature data and header structure. The header contains the root hash and block stream parameters. The root hash is the hashsum of the first block data and hashsum value. The hashsum fields in the stream form a chain, where the root hash and hash of next block fields in the actual blocks form the links of the chain.

The complete header data including the root hash is signed, and the signature is stored after the header, before the start of the first block.

3.2 Image Verification

In order to verify the integrity of the start of an SBS image, only the header data must be read and its signature checked. It is not necessary to read the entire image to establish data origin

authentication, as is done in the vast majority of current signature verification schemes. If the signature is valid, the root hash designates the hashsum of the next block, which is block one in this case. This establishes the first link in the chain of blocks.

The remaining blocks are checked while data is read. An SBS implementation reading the blocks can now check the hash of the next block before handing data to an upper layer for consumption. It can detect whether the data has been tampered with by checking all the hashsums along the chain.

3.3 Header

The header of the current protocol revision with version magic `0xe6019598` is specified in figure 3. The header fields have the following properties:

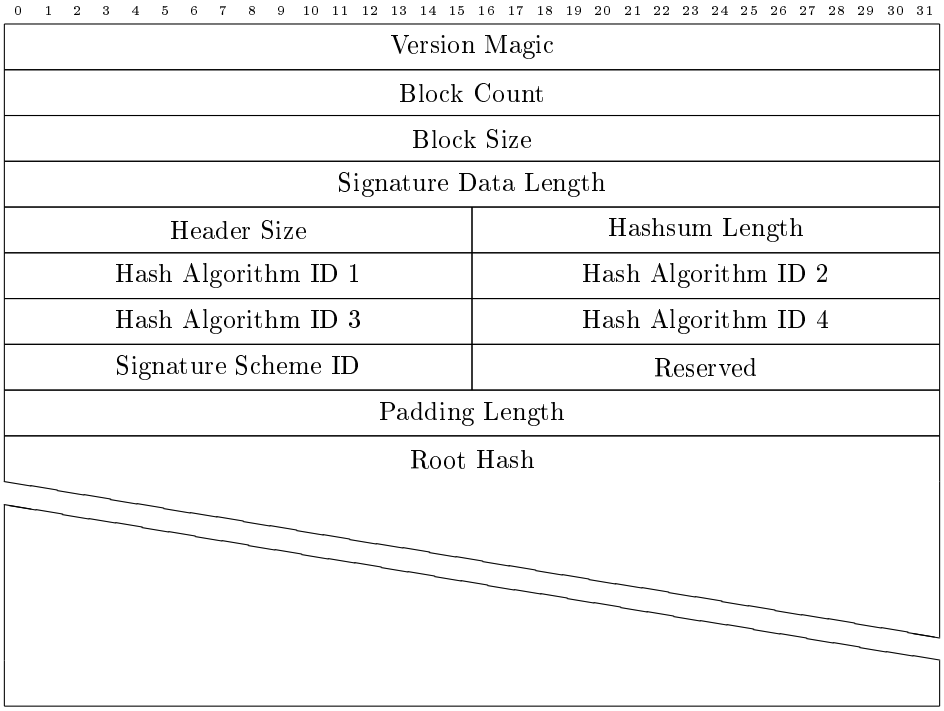


Figure 3: SBS header

- Sizes are in bytes
- Byte ordering is little-endian
- The `Version Magic` field defines the version of the SBS protocol. If an implementation does not recognize the specified magic, an error must be raised and all data discarded
- The size of the blocks is specified by the `Block Size` field. This value denotes the size of the block data plus the block hashsum field
- The size of the header is specified in the `Header Size` field and encompasses all header data from the first byte of the `Version Magic` to the last byte of the `Root Hash` field

- An implementation must check whether the `Header Size` value matches the expected value and abort if not
- The `Padding Length` field defines the number of padding bytes in the data of the first block in the stream
- The length of the `Root Hash` field is determined by the configured hash algorithms
- `Root Hash` is the concatenation of the hashsums produced by the algorithms given in the four 16-bit `Hash Algorithm ID` fields, starting with the hashsum produced by the algorithm given as `Hash Algorithm ID 1` (at the lowest order byte)
- The `Root Hash` hashsum value is the hash of the complete first block content: the hashsum and data fields, including padding bytes
- An implementation must raise an error and abort if the specified `Hashsum Length` does not match the expected length of the concatenated hashsums formed by `Hash Algorithm ID 1 .. 4`
- At least `Hash Algorithm ID 1` must be set
- A hash algorithm ID of zero designates *None*
- An implementation must raise an error and abort if it encounters an unknown `Hash Algorithm ID` or `Signature Scheme ID`
- An implementation must raise an error and abort if the `Signature Data Length` value does not match the expected value given by the configured (and supported) signature scheme.

The current protocol version specifies the hash algorithms and signature schemes as listed in tables 3 and 4.

ID	Algorithm
0	*None*
1	SHA1
2	SHA2_256
3	SHA2_384
4	SHA2_512
5	RIPEMD_160

Table 3: SBS hash algorithm IDs

ID	Scheme
1	PGP Signature (RFC 4880, section 5.2)

Table 4: SBS signature scheme IDs

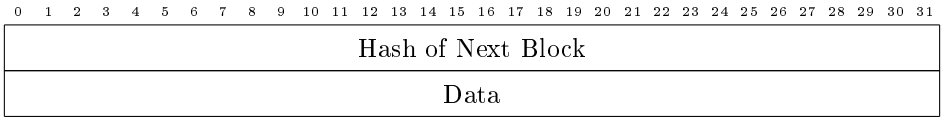
The number range `60000 .. 65535` is reserved for vendor specific algorithms and mechanisms, which are not part of the official SBS specification. It is the responsibility of the person choosing the custom algorithm ID to assure its uniqueness in the respective context, in spite of the absence of any central registry for IDs. An SBS implementation is still required to raise an error and abort if it encounters an unknown ID in this range.

3.4 Signature

The signature data following the header contains the signature produced by the given signature scheme over the complete SBS header data.

3.5 Blocks

Blocks in the stream have the following structure:



Both fields are variable in length, depending on the configured block size and hash algorithms:

$$Block\_Data\_Length = Header.Block\_Size - Header.Hashsum\_Length \tag{1}$$

The `Hash of Next Block` field is the concatenation of the hashsums produced by the algorithms given in the four 16-bit `Hash Algorithm ID` fields, starting with the hashsum produced by the algorithm given as `Hash Algorithm ID 1` (at the lowest order byte). The length of the field is given by:

$$\begin{aligned} Header.Hashsum\_Length = {} & Len(Hash\_Algo\_1) \\ & + Len(Hash\_Algo\_2) \\ & + Len(Hash\_Algo\_3) \\ & + Len(Hash\_Algo\_4) \end{aligned} \tag{2}$$

The original size of the encoded file can be calculated using the following formula:

$$\begin{aligned} Encoded\_File\_Size = {} & Header.Block\_Count * Block\_Data\_Length \\ & - Header.Padding\_Length \end{aligned} \tag{3}$$

3.6 Hash Concatenation

SBS supports the configuration of up to four hash algorithms to calculate the next block and root hashsum fields. The resulting hash value is a concatenation of all configured hashsums, as illustrated with exemplary hashsum lengths in figure 4.

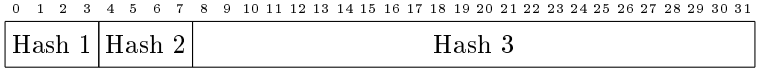


Figure 4: Concatenation of hashes with three configured hash algorithms

3.7 Padding

The first block may require padding with zero-bytes of *Padding Length*, in order to split the data evenly into blocks. Figure 5 gives an example of a signed block stream with exemplary data lengths to illustrate the basic layout.

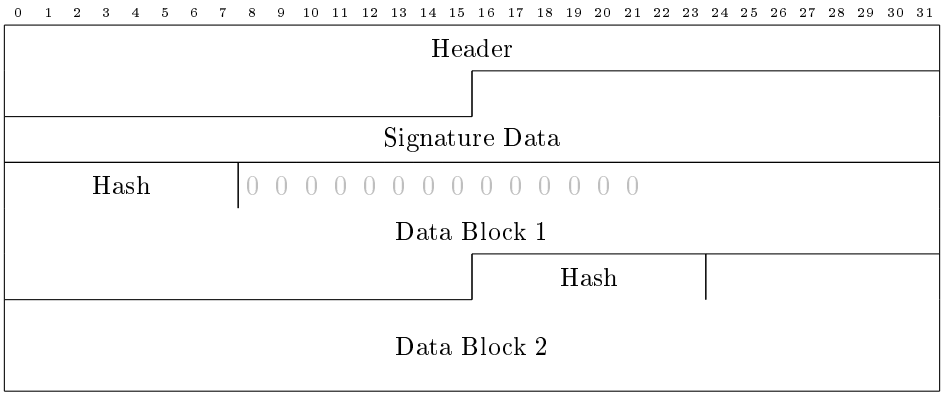


Figure 5: Example SBS stream with block 1 padding zero-bytes in gray

4 Reference Implementation

The SBS tools project[1] provides an implementation of an SBS generator, which produces signed block stream images with SHA512 hashsums and PGP signatures. The project also provides a tool to inspect SBS images.

The $\tau 0$ System Resource Manager of the Muen Separation Kernel (SK) project[2] generates system images in CSL format. Muen uses both the SBS and CSL protocols to securely deploy system images on various hardware targets.

The GRUB2 bootloader has been extended with CSL and SBS modules to process signed block and command streams[3].

Coreboot's[4] FILO bootloader payload has been extended with a CSL module to process command streams[5]. Since FILO (and libpayload) do not have cryptographic primitives (except SHA1), an implementation of SBS is not yet available in FILO.

[1] `https://www.codelabs.ch/sbs-tools`
[2] `https://muen.sk`
[3] `https://github.com/codelabs-ch/grub2`
[4] `https://www.coreboot.org/`
[5] `https://github.com/codelabs-ch/filo`