Firmware Obfuscation

Default Hash Function

The default hash function is defined as the following C# function:

```
static byte[] Hash(byte[] msg)
{
    UInt128 state = 1;
    byte[] block;
    int i;

    // Create hash value (compression function)
    for (i = 0; i < msg.Length; i++) state += (state << 8) + (msg[i] + 1);
    block = BitConverterEx.GetBytes(state);
    if (BitConverter.IsLittleEndian) Array.Reverse(block);

    // Diffuse hash value (kind of)
    for (i = block.Length-2; i >= 0; i--) block[i] ^= block[i + 1];
    return block;
}
```

The internal state s_i of the compression function is defined with its previous value s_{i-1} and the current message byte M_i as follows:

$$s_i = \left(\left(\underbrace{257}_{\text{prime number } p} \cdot s_{i-1} \right) + \left(\underbrace{M_i + 1}_{1 \ldots 256 < p} \right) \right) \bmod 2^{128}$$

General Message Authentication Code

A number message *num*(*i*), which can be represented with the bytes i_3, i_2, i_1, and i_0 with $i = i_3 \cdot 256^3 + i_2 \cdot 256^2 + i_1 \cdot 256 + i_0$, is defined as the byte sequence (i_0) for $i < 255$, or (255, i_0, i_1) for $i < 65535$, or (255, 255, 255, i_0, i_1, i_2, i_3) otherwise.

A length extension method is defined as follows:

length(*M*) = length in bytes of message *M*

len(*M*) = *num*(*length*(*M*)) || *M*

The helper function $MACS_K$(s, …) is defined as follows:

$MACS_K$(*s*, …) = *hash*(0 || *hash*(*num*(*s*) || *len*(*K*) || *len*(*N*) || *num*(*t*) || *len*(*M*))),
for *length*(*N*) > 0

or

$MACS_K$(*s*, …) = *hash*(0 || *hash*(*num*(*s*) || *len*(*K*) || 0 || *num*(*t*) || *num*(*d*))),
for *length*(*N*) = 0

with

hash function *hash*(*M*) of message *M*,
zero byte 0,
segment number *s* starting with 1,
key *K*,
nonce *N*,
type *t* (or *t* = message number *i*),
message *M*,
device number *d*, or *d* = 0 for a missing device number

The general *MAC* (= message authentication code) is defined as follows:

$$MAC_K(args) = truncate_l(\ MACS_K(1, args) \ ||\ MACS_K(2, args) \ ||\ MACS_K(3, args) \ ||\ \dots\ )$$

with

list of arguments *args*

$truncate_l(M)$ = truncate the message *M* after *l* bits, or after 128 bits if *l* is zero

Missing function arguments are taken either as byte sequences of size 0, or as the value 0.

MAC as Block Permutation

If the default hash function is used, the 128-bit block permutation *Permutation128* is allowed to be implemented with a hash function as follows:

$$Permutation128_K(P) = MAC_K(N{=}P,\ l{=}8{\cdot}length(hash))$$

If a more secure hash function than the default hash function is used, the 128-bit block permutation *Permutation128* is allowed to be implemented with a hash function as follows:

$$Permutation128_K(M) = hash(\ K \ ||\ M\ )$$

Firmware Obfuscation (Generic Algorithm)

The firmware, including DRM, is defined as a sequence of blocks M_i (for i = 0, 1, ..., n-1). A 128-bit block permutation (or a hash function) *Permutation128* is chosen for the obfuscation, which is run inside the CTR mode of operation with a 64-bit *nonce*, a 32-bit message index *i*, and a 32-bit *counter*. Index *i* and *counter* are starting with zero and use the little Endian format. The *counter* is incremented by 1 for the next block.

$$C_i = Permutation128\text{-}\mathrm{CTR}_K^N(M_i) \quad \text{with } N = \text{32-bit-}counter \ ||\ \text{32-bit-index-}i \ ||\ \text{64-bit-}nonce$$

Firmware Obfuscation (with MAC)

The firmware, including DRM, is defined as a sequence of blocks M_i (for i = 0, 1, ..., n-1). The obfuscation is defined by:

$$C_i = MAC_K(N{=}nonce,\ t{=}i,\ l{=}8{\cdot}length(M_i)) \oplus M_i$$

with a 64-bit *nonce*.

TSF Stream Format (Light Version)

A tagged stream format is a binary stream representation with a *NUL* terminated collection of nested, sorted and unambiguously tagged data items. The data of a missing item is set to zero (or an empty string).

A *flexible number* is defined with a single byte. If this byte equals $0ff_{16}$, then the next two bytes represent the flexible number. If these next two bytes equal $0ffff_{16}$, then the next four bytes represent the flexible number, and so on. Flexible numbers are prefixed with *num*.

An *id* represents a strictly monotonically increasing non-zero identification number within a *NUL* terminated object collection sequence. The *id* with number 31 is reserved.

Identified objects consist of an *id* byte and object data. An *id* byte consists of the *id* field (higher significant 5 bits) and of the *type* field (lower significant 3 bits):

id = 0	type = 0	Zero tag
id = 0	type = 1	4-byte header (indicating little Endian): 01_{16} $e1_{16}$ 74_{16} 73_{16}
id = 0	type > 1	Reserved
id > 0	type = 0	Data size: 1 byte
id > 0	type = 1	Data size: 2 bytes
id > 0	type = 2	Data size: 4 bytes
id > 0	type = 3	Data size: 8 bytes
id > 0	type = 4	Data size is defined by *numDataSize*; subsequent items: <*numDataSize*>
id > 0	type = 5	Object collection vector of length 1 until zero tag; subsequent items: <*objs*[0]>
id > 0	type = 6	Object collection vector; subsequent items: <*numCount*><*objs*[0]><*objs*[1]>...
id > 0	type = 7	Reserved

The last zero tag may be omitted for TSF fragments.

Used Data Types

BYTE *array*[]	Sequence of bytes (whose length is given by the stream)
BYTE0 *array*[]	Sequence of bytes (whose length is given by the stream). This entry must not be omitted if it contains only zeros.
UINT *uint*	Unsigned integer (whose size is given by the stream)
Collection[]	A missing data type marks an identified object collection. Postfixed void brackets mark a vector (whose size is given by the stream) of *NUL* terminated identified object collections.

Documentation

A document with TSF format is described as follows:

#1	#2	#3	#4	Object name	Description

#1: Id at lowest level (root node is reachable via one leaf)
#2: Id of an embedded object (root node is reachable via two leaves)
#3: Id of an embedded object (root node is reachable via three leaves)
#4: Id of an embedded object (root node is reachable via four leaves)

Composition of the First Obfuscated Sequence Block (*Nonce* || T_0 || C_0)

BYTE *Nonce*[8]	64-bit nonce
BYTE T_0[*TLen*]	Authenticated message digest of C_0 (with default value for *TLen* = `MIN(sizeof(`*hash*`) / 2, 16)`
BYTE C_0[]	Obfuscated TSF fragment (= first obfuscated sequence block M_0)

C_0: Obfuscated TSF fragment without header in little Endian format M_0.
[Assumption: Header = 01_{16} $e1_{16}$ 74_{16} 73_{16} / Id = 134 / Version = 0.0]

2				UINT Signature	= 0x0b7c1f9a (optional)
3				Info	
	3			UINT Id	
	5			UINT Features	
	7			UINT MajorVersion	DWORD dwVersion = (MajorVersion << 24) + (MinorVersion<<16) + Revision;
	9			UINT MinorVersion	
	11			UINT Revision	
	13			UINT Build	
	15			UINT Config	
	17			UINT Date	(= UnixTime)
	25			BYTE Description[]	
	27			BYTE Name[]	
	29			BYTE Version[]	Version string (with "V" at the beginning)
5				Drm[]	DRM single test (which all have to be satisfied)
	3			UINT Type	Type: 0 – Manufacturer Id 1 – Hardware Id 2 – Hardware Version 3 – Internal Device Number 4 – Device Number 5 – Customer Id 6 – Manufacturing Date (= UnixTime) 64 – Hardware Features
	5			UINT Min	MinMax.Min = Min
	7			UINT Max	MinMax.Max = Max
	9			UINT MinMax	MinMax.Min = MinMax.Max = MinMax
	11			UINT XorMask	Features.XorMask = XorMask
	13			UINT ReqFeatures	Features.ReqFeatures = ReqFeatures
7				UINT EraseRegions	Bit mask: 1 – Clear total memory area 2 – Clear application memory area

					4 – Clear data memory area
9				EraseList[]	Regions may not overlap and must be sorted upwards. They must not include the bootloader area. Regions of zero size will be ignored.
	3			UINT StartAddress	
	5			UINT Length	
15				MemoryArea[]	Regions may not overlap and must be sorted upwards. They must not include the bootloader area. Regions of zero size will be ignored.
	17			UINT StartAddress	
	19			BYTE0 Data[]	

DRM single test of type *x* (e.g. *x* = Hardware Id):

```
drmTestFailed =
  (MinMax.Min > 0 && x < MinMax.Min) ||
  (MinMax.Max > 0 && x > MinMax.Max) ||
  (((x ^ Features.XorMask) & Features.ReqFeatures) !=
  Features.ReqFeatures);
```

Composition of the Next Obfuscated Sequence Blocks (T_i || C_i)

BYTE T_i[*TLen*]	Authenticated message digest of C_i
BYTE C_i[]	Obfuscated TSF fragment (= obfucsated sequence block M_i)

C_i: Obfuscated TSF fragment without header in little Endian format M_i.
[Assumption: Header = 01_{16} $e1_{16}$ 74_{16} 73_{16} / Id = 134 / Version = 0.0]

15				MemoryArea[]	(see above)
	17			UINT StartAddress	
	19			BYTE0 Data[]	

Composition of the First Unobfuscated Data Block (H_0 || D_0)

BYTE H_0[*HLen*]	Checksum of D_0 with *HLen* = 4
BYTE D_0[]	TSF fragment (= data block D_0)

D_0: TSF fragment without header in little Endian format.
[Assumption: Header = 01_{16} $e1_{16}$ 74_{16} 73_{16} / Id = 134 / Version = 0.0]

2				UINT32 Signature	= 0x0b7c1f9a (optional, see above)
3				Info	(see above)
5				Drm[]	See above; additional values for *type*: 129 – Application Id 130 – Application Version 131 – Application Build 132 – Application Config 133 – Application Date (= UnixTime) 192 – Application Features
7				UINT EraseRegions	8 – Clear configuration memory area 16 – Clear configuration data memory area

9				EraseList[]	(see above; limited by info area)
15				MemoryArea[]	(see above; limited by info area)

Composition of the Next Unobfuscated Data Blocks (H_k || D_k)

BYTE H_k[*HLen*]	Checksum of D_k
BYTE D_k[]	TSF fragment (= data block D_k)

D_k: TSF fragment without header in little Endian format.
[Assumption: Header = 01_{16} $e1_{16}$ 74_{16} 73_{16} / Id = 134 / Version = 0.0]

15				MemoryArea[]	(see above)
	17			UINT StartAddress	
	19			BYTE0 Data[]	

Firmware File Format

TSF stream in little Endian format with 4-byte header: 01_{16} $e1_{16}$ 74_{16} 73_{16}.

1				StreamInfo	
	2			UINT Id	Id = 134
27				Firmware[]	
	3			UINT Salt	Unique UTC time in seconds since January 1, 1970
	5			BYTE0 ObfuscatedInfo[]	No obfuscation if the *Salt* value is missing; the composition is that of M_0;
	6			Keys	
		3		BYTE0 ObfuscationKey[]	
	7			ObfuscatedBlocks[]	
		3		BYTE0 Block[]	
	9			UINT ConfigurationType	0 – first matching configuration 1 – no configuration 2 – index based configuration: *Configuration*[*ConfigurationIndex*]
	11			UINT ConfigurationIndex	
29				Configuration[]	
	25			Sequence[]	
		5		Drm[]	
			3	UINT Type	
			5	UINT Min	
			7	UINT Max	
			9	UINT MinMax	
			11	UINT XorMask	

			13	UINT ReqFeatures	
		7		UINT EraseRegions	
		9		EraseList[]	
			3	UINT StartAddress	
			5	UINT Length	
		15		MemoryArea[]	
			17	UINT StartAddress	
			19	BYTE0 Data[]	
30				UINT Checksum	Checksum from tag #1 until tag #29
0					

Obfuscation for TSF Item “ObfuscatedInfo”

ObfuscatedInfo may optionally be obfuscated. In this case, *ObfuscatedInfo* can be obfuscated (and deobfuscated) with the following C# functions:

```
static void AddObfuscation(byte[] data, uint salt)
{
    byte[] reference = (byte[])data.Clone();
    uint state = (uint)(3000001321 * salt + 9000000101);
    data[0] ^= (byte)state;
    for (int i = 1; i < data.Length; i++)
    {
        state = (uint)(3000001321 * (state + reference[i-1]) + 9000000101);
        data[i] ^= (byte)(state / 2);
    }
}

static void RemoveObfuscation(byte[] data, uint salt)
{
    uint state = (uint)(3000001321 * salt + 9000000101);
    data[0] ^= (byte)state;
    for (int i = 1; i < data.Length; i++)
    {
        state = (uint)(3000001321 * (state + data[i-1]) + 9000000101);
        data[i] ^= (byte)(state / 2);
    }
}

// RemoveObfuscation:
//
// salt:      1
// data in:   bf 4c 52 d1 ab 09 ab 92  44
// data out:  31 32 33 34 35 36 37 38  39  (= "123456789")
```

Checksum Algorithm

The 32-bit checksum can be obtained with the following C# function:

```
static public uint ComputeChecksum(byte[] arr, int offset, int len)
{
    uint state = 1;

    for (int i = 0; i < len; i++)
    {
        state += (uint)((state << 8) + (arr[offset + i] + 1));
    }
```

```
        return state;
    }
```

Tiny Bootloader

The bootloader can be compiled with the *tiny bootloader* option. In this case, the TSF library will not be included in the bootloader. Therefore, the obfuscated blocks C_i and D_i cannot be in TSF format. They are instead constructed as follows (by using little Endian byte order):

C_0 or D_0:

UINT32 Signature	= 0xf8139cbd
UINT32 EraseRegions	
UINT32 Count	Number of DrmItems (1...21)
DrmItems[Count]	At least one element with *Type* == 1 (= *hardwareId*) is required

DrmItems for (*type* & 0x40) == 0:

UINT32 Type	
UINT32 Min	
UINT32 Max	

DrmItems for (*type* & 0x40) != 0:

UINT32 Type	
UINT32 XorMask	
UINT32 FeatureRequest	

C_i or D_i (for $i > 0$):

UINT32 Address	
UINT32 Count	= 1...256
BYTE Data[Count]	