BI.ZONE Follow
Jun 8, 2021 · 26 min read · Listen

Measured Boot and Malware Signatures: exploring two vulnerabilities found in the Windows loader

By Maxim Suhanov, 2021

Introduction to Boot Security

There are two major concepts for boot security: verified boot and measured boot.

The verified boot process ensures that components not digitally signed by a trusted party are not executed during the boot. This process is implemented as Secure Boot, a feature that blocks unsigned, not properly signed, and revoked boot components (like boot managers and firmware drivers) from being executed on a machine.

The measured boot process records every component before executing it during the boot, these records are kept in a tamper-proof way. This process is implemented using a Trusted Platform Module (TPM), which is used to store hashes of firmware and critical operating system (OS) components in a way that forbids changing these hashes to values chosen by a malicious program (later, these hashes could be signed and sent to a remote system for health attestation).

Both concepts can be implemented and used either separately or simultaneously. Some technical details about these concepts and their Windows implementation can be found in other sources [1][2].

Four more concepts are known: post-boot verification, booting from read-only media, booting from a read-only volume (image), and pre-boot verification.

Post-boot verification is performed by a program launched after the boot process has been completed or at its late stages. Such a program verifies the integrity of previously executed operating system components and their configuration data. Obviously, malware already running at a higher privilege level can completely hide itself from this type of verification: for example, by intercepting file read requests and controlling file data returned to the requesting program.

Still, it could be used as a defense-in-depth measure by anti-malware software, endpoint detection and response solutions, and cryptographic components.

Booting from read-only media is sometimes used to get an immutable, known good environment, while assuming the safety of an entire pre-OS environment, which includes Basic Input/Output System (BIOS) and Unified Extensible Firmware Interface (UEFI). If a malicious program is embedded into a pre-OS environment, the whole approach gets compromised.

The usage of live distributions for online banking was proposed many years ago [3]. Currently, there are similar proposals for using live distributions to enhance security when working from home. The concept performs perfectly against “traditional” malware, thus protecting against most (but not all) malware-related threats.

It’s also possible to trick an operating system booting from removable media into automatic (requiring no user interaction) execution of code stored on attached non-removable media. This could be achieved during the transition to own (native) storage drivers from BIOS/UEFI functionality used to read data from a boot drive. For more details about this and similar code execution issues, see my past work [4].

Booting from a read-only volume (image) is a similar concept, but only immutable operating system files are stored in a read-only volume (image). Technically, this volume (image) can be modified, but its integrity is validated using a hash or a hash tree. This hash or the root hash of this tree is signed and validated by a hardware-protected root of trust (so, this approach is linked to an existing verified boot implementation). More technical details can be found in another source [5].

Pre-boot verification is a lesser-known technique[1]. Typically, it is implemented as a Peripheral Component Interconnect (PCI) device or as a custom UEFI image. Before launching a boot loader found on a boot drive, the verification process checks the integrity of hardware configuration, exposed firmware memory, executable files, and other data (like registry keys and values of a Windows installation).

Originally, this was implemented as a PCI device with option read-only memory (option ROM) containing initialization (pre-OS) code that installs a breakpoint at the location of the first instruction of a boot loader. When this breakpoint is hit, a custom operating system is launched from the memory of that PCI device. Then, this custom operating system parses file systems found on the attached drives and performs the verification process (a list of known good hashes is stored in the memory of the PCI device or on a system drive with its integrity verified). If the verification succeeds, the original boot loader gets executed, launching the verified operating system.

Current implementations move away from PCI devices to providing custom UEFI images or shipping motherboards with such custom UEFI images. These custom UEFI images contain parsers necessary for the verification process. Additionally, there is one implementation based on a USB drive, which must be configured as a boot device (instead of a drive to be verified, so there is no need to break into the boot process using a breakpoint or an addon to the BIOS/UEFI environment, assuming that a user won’t change the boot order).

This approach has limited capabilities to verify the integrity of firmware, including BIOS and UEFI, but the usage of a custom UEFI image moves the trust boundary, including most firmware into the area to be trusted by security design.

Another shortcoming is that it is nearly impossible to produce the same file system and Windows registry parsing code as found in the official Windows implementations. Typically, when Linux users mount an NTFS file system, they see the same directory layout and the same file contents as Windows users exploring exactly the same file system. However, there are edge cases and file system drivers that can behave differently, giving back different data (while the raw data, which is actually stored on a drive, is the same). This also applies to the official and third-party Windows registry implementations.

For example, most pre-boot verification products have no support for Windows registry transaction log files. This means that malware running at the kernel level can introduce modifications to registry keys and values that would be unnoticed by the pre-boot verification (because transaction log files are simply ignored). Since the Windows kernel fully supports transaction log files, these changes will be visible to the operating system during and after the boot. This attack has been explained in detail in my past work [6].

The verified and measured boot processes are expected to be immune to such attacks because they verify and measure code and data just before using them. If, for any reason, a file to be executed produces different hashes depending on a file system driver used to read that file, this would not affect security because the file is verified/measured and then executed using the same driver (a hash computed during the verification/measurement will match the file contents read for its execution). In other words, the same implementation is utilized to verify/measure data and then, immediately, use it (technically speaking, this could be the same memory buffer used to verify/measure data and use it).

But this is not always the case!

This paper will focus on two vulnerabilities discovered in the measured boot implementation found in the Windows loader (winload.exe or winload.efi), both highlight a longstanding problem of validating data and code in separate software components.

[1]Unless you live in Russia, where trusted boot modules described here are required by certain regulatory bodies for multiple usage scenarios.

Early Launch Anti-Malware

Before moving to the vulnerabilities, let us take a look at an early anti-malware interface provided by the Windows kernel (ntoskrnl.exe).

If two major boot security concepts, verified boot and measured boot, work as expected and intended, and no hardware/firmware vulnerabilities are considered, the earliest insertion point for malware is a boot-start driver.

To combat malicious boot-start drivers, Microsoft introduced an interface called Early Launch Anti-Malware (ELAM). This happened in Windows 8 [7].

A Microsoft-signed ELAM driver starts before other drivers and validates them as well as their linked dependencies. For each boot-start driver, an ELAM driver can return the following values:

- This driver is unknown (not identified as good or bad);
- This driver is good (not malicious);
- This driver is bad (malicious);
- This driver is bad (malicious) but critical for the boot process (the operating system won't boot without this driver).

Additionally, an ELAM driver can install a callback to record registry operations performed by boot-start drivers. These records can be sent to a runtime anti-malware component.

Anti-malware software using an own ELAM driver is allowed to run as a protected service. Such a service is given code integrity protections, this feature was introduced in Windows 8.1 [8].

More than one ELAM driver can be active during the boot process. Each ELAM driver checks a given boot-start driver independently and a final decision for this driver is based on the following scores (ranks):

Decision	*Score (rank)*
Unknown	0
Good	1
Bad	3
Bad but critical	2

Table 1. ELAM scores (ranks)

A decision with a higher score (rank) wins. So, if one ELAM driver returns "good" and another ELAM driver returns "bad", the final decision for this boot-start driver is "bad".

Based on this decision and a policy, the kernel allows or denies a boot-start driver. The following policies can be configured:

- All drivers are allowed;
- Only good, unknown, and bad but critical drivers are allowed (by default);
- Only good and unknown drivers are allowed;
- Only good drivers are allowed.

The following data can be utilized by an ELAM driver to make a decision:

- The path to the driver file;
- The registry path to a corresponding service entry;

- Certificate information for the driver (a publisher, an issuer, and a thumbprint);
- The image (file) hash.

Importantly, file contents are not exposed to an ELAM driver, so there is no way to apply traditional malware signatures (when the principle of Dynamic Root of Trust for Measurement, DRTM, is applied, it would be impossible to read anything from a drive before relevant drivers have been initialized, because the firmware functionality previously used to read boot-start drivers into the memory is untrusted [9]). However, blocking malicious boot-start drivers by their paths, hashes, and certificates is expected to be effective.

Microsoft defined some additional rules:

1. An ELAM driver is given a limited time to check a single boot-start driver;
2. An ELAM driver is given a limited time to check all boot-start drivers;
3. An ELAM driver is given a limited amount of memory for its code and configuration data;
4. An ELAM driver must store its signatures in an ELAM registry hive (*C:\Windows\System32\config\ELAM*), under a specific registry key (named after an anti-malware vendor);
5. An ELAM driver must validate its signatures;
6. An ELAM driver must handle invalid signatures (in this case, it should treat all boot-start drivers as unknown);
7. An ELAM driver should revoke the attestation (invalidate the measured boot state) when a malicious boot-start driver (or another policy violation) is identified.

When the ELAM interface first appeared, it was believed to be nearly useless [10]. A bootkit could completely bypass an ELAM driver by writing its malicious code into a volume boot record (VBR) or by replacing an initial program loader (IPL), both are executed before the Windows kernel and, thus, before the ELAM interface. But the proliferation of verified/measured boot raised the bar, you can no longer say the ELAM interface is nearly useless.

ELAM and Measured Signatures

During measured boot, ELAM signatures are read and then measured by the Windows loader [11]. This puts ELAM signatures into the chain of trust, so missing or downgraded signatures can be detected and reported during the attestation.

Not all possible locations of ELAM signatures are measured, but only those stored in specific registry values within the ELAM hive (these values are called “Measured”, “Policy”, and “Config”, all of them can contain vendor-specific data; registry values with different names or with types other than REG_BINARY are not measured).

Table 2. Signature data stored by Trend Micro and Microsoft (respectively), measured values marked with bold

It is important to mention that ELAM signatures are measured by the Windows loader, but the ELAM interface is provided by the Windows kernel.

Real-World ELAM Drivers

In February 2021, I reverse-engineered ELAM drivers shipped with popular anti-malware products (those without ELAM drivers were out of scope). In total 26 products having 25 unique ELAM drivers (two products share exactly the same ELAM driver), with 24 installed in the default configuration (one product writes its ELAM driver to a system volume, but no corresponding registry entry is created, the reason is unclear).

Interestingly, all ELAM drivers examined had embedded certificate information required for launching a protected anti-malware service, but most of them (15 out of 26) do not do any checks against boot-start drivers and either return a hard-coded decision (11 out of those 15) or do not provide decisions at all (4 out of those 15). These ELAM drivers will be referred to as placeholder drivers. Two ELAM drivers record boot-start driver information for a runtime anti-malware component (but no actual checks are performed in the ELAM drivers, the decision is always “unknown”). One ELAM driver reports all boot-start drivers as “good” (with no actual checks performed).

11 ELAM drivers perform at least some checks against boot-start drivers (the exact nature of these checks and the number of existing malware signatures were out of scope). One of them has a hard-coded list of known good certificates (the ELAM hive is not used to store signature data), two use both the ELAM hive and the SYSTEM hive for signature data, one uses the ELAM hive and then the SYSTEM hive as a fallback for signature data, seven read signature data from the ELAM hive only.

Only two (out of those 11) ELAM drivers can revoke the attestation, others never call a corresponding routine.

Now, let us mention some of the names.

Surprisingly, the ELAM driver shipped with Windows Defender does not follow all of the rules: it uses the SYSTEM hive as a fallback location for signature data (besides the ELAM hive, which is a primary location) and it does not revoke the attestation when a malicious boot-start driver is detected.

The ELAM driver shipped with Kaspersky and ZoneAlarm products (ZoneAlarm uses exactly the same ELAM driver as made by Kaspersky) can read signature data from two locations: the ELAM hive and the SYSTEM hive. And it does not revoke the attestation too.

The ELAM driver shipped with Sophos products simply marks all boot-start drivers as “good”. This was reported as a security issue to Sophos, but they consider it as an intended feature, which does not weaken the security, see Appendix I.

Detailed results can be found in Appendix II.

To summarize, most anti-malware products do not use their ELAM drivers to scan for malicious boot-start drivers. Instead, they utilize ELAM drivers to launch themselves as protected services and the core ELAM functionality is limited to either providing a single hard-coded decision or no decisions at all.

ELAM Hive

The ELAM hive is stored at this location: *C:\Windows\System32\config\ELAM*.

This a registry file, its binary format has been fully described in my previous work [12]. Readers are encouraged to make themselves familiar with this format first.

There are several key points required to understand the vulnerabilities:

1. A key node (a binary structure used to describe a single registry key) can point to a subkeys list (which is a list of offsets to key nodes describing subkeys of this registry key);
2. Similarly, a key node can point to a values list (a list of offsets to key values, each key value describes a single registry value belonging to this registry key);
3. An offset equal to 0xFFFFFFFF does not point anywhere (this value is used to express “nil”);
4. Such offsets are not absolute, one needs to add 4096 bytes to get an offset from the beginning of a registry file (and each structure is preceded with the four-byte size field, it is a cell header, and cell data is a structure itself);
5. A key node and a key value store a name of this registry key and a name of this registry value respectively, this could be either an extended ASCII (Latin-1) string or an UTF-16LE string (name strings that can be stored as extended ASCII strings are compressed into this form);
6. A subkeys list must be sorted by an uppercase name of a subkey (the lexicographical order) in order to enable case-insensitive binary search across subkeys;
7. On the other hand, a values list is not required to be sorted;
8. A key value records the data type of this registry value (for example, REG_BINARY) and points to value data;
9. Value data not larger than four bytes is stored directly in a key value;
10. Value data larger than four bytes is stored at a different offset, this offset is recorded in a key value;
11. Value data larger than 16344 bytes is stored in segments of 16344 bytes or less (for the last segment of value data), offsets to these segments are referenced in a list, an offset to this list is stored in a big data record, an offset to this record is stored in a key value (this applies to the hive format versions 1.4, 1.5, and 1.6, previous versions store value data as described previously, without using segments).

An example walk-through of a registry file is below:

Fig. 1. A root key node of the ELAM hive (shown as selected): green — the number of subkeys (2), red — the offset to a subkeys list (0×3328, the absolute offset is 0×3328+4096=0×4328), yellow — the key name (“ROOT”, it’s an ASCII string)

Fig. 2. A subkeys list for a root key (shown as selected): yellow — the number of elements in this list (2), red — the offsets to two subkeys (0×3188 and 0×0120; in this type of subkeys list, four bytes after each offset contain a name hash used to speed up lookups), note that elements are stored in the sorted order (0×3188 corresponds to a key node called “Trend Micro”, 0×0120 corresponds to a key node called “Windows Defender”)

Fig. 3. A key node called “Windows Defender” (shown as selected): red — the number of values (1), yellow — the offset to a values list (0×3180)

Fig. 4. A values list (shown as selected): the only item is 0×0230

Fig. 5. A key value (shown as selected): yellow — the value data size (0×215C, or 8540 bytes), red — the value data offset (0×1020; this field would store value data directly if it is four bytes or less), blue — the value type (3 or REG_BINARY), green — the value name (“Measured”, it’s an ASCII string)

Fig. 6. Value data (shown as selected)

Fig. 7. When value data is larger than 16344 bytes and the hive format version is not less than 1.4, the offset field underlined in red in Fig. 5 points to this structure (shown as selected), it is called “big data”: red — the number of value data segments (2), yellow — the offset to a list of segments (0×3188)

Fig. 8. A list of value data segments (shown as selected): red — two offsets (0×4020 and 0×8020)

Fig. 9. A segment (shown as selected), the first one contains 16344 bytes, the last one contains remaining value data (there are two segments only as shown in Fig. 8)

There are several implementation details worth noting:

1. When a registry hive is mounted (loaded), it is checked for format violations. When a format violation is detected, an attempt is made to correct it or to delete a related registry structure, including references to this structure (this decision is based on what exactly is wrong);
2. In particular, the lexicographical order of elements in all subkeys lists is checked. If a comparison of two subkeys, the current key and the preceding one in a given list, reveals that they are in the wrong order, the current key is deleted;
3. Usually, when a hive is mounted, usermode applications can not write to its underlying file because it is locked. When an operating system has finished the boot, the ELAM hive is kept unmounted. This is for performance reasons;
4. The ELAM hive is using the format version 1.5. So, big data records can be encountered in this hive;
5. During the early boot, the Windows loader reads the ELAM hive into a single chunk of memory.

Measured Boot Vulnerabilities

In late 2020 and early 2021, I discovered and reported two vulnerabilities that allow a malicious program running with administrator privileges to corrupt, downgrade, or delete ELAM signatures without affecting the measured boot process.

In particular, the Windows loader measures expected registry values in the ELAM hive, while the Windows kernel sees different registry data in that hive, containing either corrupt or downgraded ELAM signatures, or having no corresponding registry values at all (thus, no ELAM signatures).

These vulnerabilities exploit differences in registry parsing code found in the Windows loader and the Windows kernel.

Table 3. Vulnerabilities discovered with their corresponding CVE IDs

Both vulnerabilities were found eligible for a bounty. For CVE-2021–27094, it took more than 90 days to deploy a fix. And this fix resulted in a data corruption issue.

CVE-2021–28447

This vulnerability is pretty straightforward.

When measuring ELAM values with data larger than 16344 bytes, the Windows loader does not parse a big data record encountered. Thus, if an ELAM blob being measured is larger than 16344 bytes, it is measured incorrectly.

This is a decompiled function used to get value data for measured ELAM values:

Fig. 10. A decompiled function used to get value data for measurements

This function has no checks for the hive format version, value data size, and no code for handling the big data record.

The function reads a cell pointed by the *KeyValue->Data* field. In usual cases, when value data is not larger than 16344 bytes, this cell would contain entire value data (thus, the *memmove()* call just moves this data into a heap variable, *Heap*).

When value data is larger than 16344, this cell would contain the big data structure (as seen in Fig. 7). The function won't parse this structure, but use it as value data (which is obviously wrong).

Since the hive is loaded into a single chunk of memory and a big data record is smaller than the expected value data size, the *memmove()* call actually copies this big data record and subsequent registry data from a loaded registry file into the heap variable.

So, the Windows loader does not measure proper value data. Instead, it measures registry file internals, starting from the big data record and going further up to the value data size.

Under specific conditions, this allows an attacker to modify ELAM blobs without affecting their measurements.

For example, if value data segments are stored before the big data record (at a lower offset within the registry file), they are not included in the *Heap* variable, which is then measured. Thus, real value data is not measured at all.

Alternatively, if value data segments are stored after the big data record, they are not measured completely (trailing value data is beyond the range starting at the big data record and going further up to the value data size). Thus, it is possible to alter trailing value data without changing the hash calculated during the measurement.

Since the ELAM hive is not loaded after the boot, it is possible to alter it in any way (for example, by using a HEX editor), thus an attacker is not bound to standard and native API calls when modifying the registry file.

Root cause

Apparently, this vulnerability was caused by legacy code lacking support for the big data record case. Previously, there was no need to read such registry values in the Windows loader.

Fix

Microsoft fixed the vulnerability by implementing the support for the big data record case in the Windows loader. This vulnerability does not affect ELAM drivers evaluated — their ELAM blobs do not reach the threshold of 16344 bytes.

Original vulnerability report

```
# Summary

When an ELAM driver stores a binary larger than 16344 bytes in one of three measured values (called "Measured", "Policy", or "Config") within the ELAM hive ("C:\Windows\System32\config\ELAM"), this binary isn't measured correctly by the Windows loader (winload.exe or winload.efi).

Under specific conditions, a modification made to an ELAM blob won't result in different PCR values, thus not affecting the measured boot (since PCR values are equal to the expected ones).

# Description

## Steps to reproduce

(Screenshots attached.)

1. Mount the ELAM hive using a registry editor.

2. Add a new key under the root of the ELAM hive. Assign a new value to this key (in this report, the value will be called "Measured").

3. Write more than 16344 bytes of data to that value (see: "01-elam-blob.png").

4. Unmount the ELAM hive.

5. Reboot the system.

6. During the boot, the Windows loader measures data starting from the beginning of the CM_BIG_DATA structure as pointed by the CM_KEY_VALUE structure describing the "Measured" value (see: "02-elam-blob-measured.png"). Since the expected data length is larger than the CM_BIG_DATA structure, subsequent bytes of the hive file (actually, from the memory region used to store the hive file loaded) are included into the measurement (instead of actual value data).

7. After the boot, change (using a registry editor) several bytes within the value data, without altering the data size (see: "03-elam-blob-altered.png").

8. Reboot the system.

9. During the boot, the Windows loader will see the same CM_BIG_DATA structure and subsequent bytes as value data (see: "04-elam-blob-altered-measured.png").

## Root cause

The Windows loader doesn't support parsing value data stored using the CM_BIG_DATA structure. This structure is used when the hive format version is 1.4 or newer and value data to be stored is larger than 16344 bytes.

The ELAM hive uses the format version 1.5. Thus, the CM_BIG_DATA structure is supported in the NT kernel, but not in the Windows loader.

The OslGetBinaryValue routine (in the Windows loader) provides back a pointer to cell data containing the CM_BIG_DATA structure instead of parsing this and related structures and then providing a pointer to consolidated data segments.

## Attack scenarios

First, ELAM blobs larger than 16344 bytes aren't measured correctly. This is a serious security issue by itself.

Finally, if an ELAM driver uses existing measured ELAM blobs larger than 16344 bytes, a malicious usermode program could alter (corrupt or downgrade) these blobs without affecting the measured boot.

Such an attack is possible when:
* a list of cells containing value data segments is stored before the CM_BIG_DATA structure, or
* such value data segments are stored before the CM_BIG_DATA structure, or
* a list of cells containing value data segments and such value data segments are all stored after the CM_BIG_DATA structure, but there is a large gap after the CM_BIG_DATA structure (which isn't smaller than the defined value data size, so the hash calculation won't reach the actual value data, or it's smaller than that, but the hash calculation doesn't reach the modified bytes of actual value data).

Under any specific condition defined above, changing offsets to value data segments or changing value data segments respectively won't be noticed during the measurement. (Since the hash is calculated over the internals of the hive file, but not over the actual value data.)

Since the ELAM hive isn't loaded after the boot, a malicious usermode program can open it and alter its data in any way possible (this is not limited to registry functions exposed by the Advapi32 library, the hive file can be opened and edited in a HEX editor), thus exploiting any pre-existing condition defined above.

## Possible solution

Handle the CM_BIG_DATA structure when parsing a registry value using the Windows loader.
```

The screenshots are attached below.


01-elam-blob.png
02-elam-blob-measured.png
03-elam-blob-altered.png

04-elam-blob-altered-measured.png

CVE-2021–27094

This vulnerability is slightly more complicated.

When loading a hive, either in the Windows loader or in the Windows kernel, it is checked for format violations. These checks are performed twice for hives loaded by the Windows loader and then passed to the Windows kernel in the memory (this includes the ELAM hive).

In the Windows loader and in the Windows kernel, these checks are similar, but not the same. In particular, the Windows loader does not check the lexicographical order of elements in subkeys lists.

Since the ELAM hive is used by an ELAM driver launched by the Windows kernel, this inserts the lexicographical order check between the measurement of ELAM blobs by the Windows loader and their usage by an ELAM driver. It is possible to exploit this check to remove a registry key containing an ELAM blob after it has been measured by the Windows loader, but before it is used by an ELAM driver.

In order to achieve this, an attacker needs to insert an empty (containing no values) key into the ELAM hive, under its root key (this could be done by mounting the hive and then using standard API calls to create a key, the hive should be unmounted before proceeding to the next step), then break the order of subkeys in a way that would force a key with a measured ELAM blob to be deleted by the Windows kernel (during the check). The last step requires an attacker to open the hive file in a HEX editor for moving the elements in a corresponding subkeys list (so this hive must be unmounted). (The same could be done automatically, of course.)

Let us take a look at the following layout of the ELAM hive:

1. Key: *Windows Defender*
   - Value: *Measured*
2. Key: *zz*
   - No values

(The order of keys reflects the order of key node offsets in a subkeys list of a root key.)

This layout is valid, it could be created using standard API calls. The attacker needs to modify the subkeys list to get the following layout:

1. Key: *zz*
   - No values
2. Key: *Windows Defender*
   - Value: *Measured*

(This could be achieved by reversing the order of two elements in a subkeys list.)

Now, the lexicographical order is broken:

Upcase(“*zz*”) > Upcase(“*Windows Defender*”)

The Windows kernel will delete the “*Windows Defender*” key before it is used by an ELAM driver.

But the Windows loader won’t notice this corruption. It will skip the “*zz*” key (it has no values to be measured) and measure the “*Measured*” value found in the “*Windows Defender*” key.

So, we have a situation when an ELAM blob is measured as expected, but it is deleted before an ELAM driver is executed, thus before this blob is used.

Root cause

This vulnerability is caused by an absent check for the lexicographical order of elements in a subkeys list. Apparently, this check has been deliberately removed from the Windows loader because Unicode key names are allowed and there is no way to do case-insensitive comparisons without a proper uppercase table (a table used to convert characters into their corresponding uppercase versions).

Since the Unicode uppercase table is used by the Windows kernel, this check is implemented there. This also requires running the checks twice (relaxed checks in the Windows loader and complete checks for previously loaded hives in the Windows kernel).

Interestingly, the current implementation requires the SYSTEM hive to be loaded before loading the NLS tables (which include the Unicode uppercase table), although this can be refactored for Unicode characters.

Fix

Microsoft fixed the vulnerability by introducing the lexicographical order check in the Windows loader. Since the Unicode uppercase table is not used by the Windows loader, this fix is limited to key names using ASCII characters only. Currently, no ELAM drivers are known for using non-ASCII characters in key names.

The fix also causes a serious data corruption issue when checking the SYSTEM hive — this hive, like the ELAM hive, is loaded during the early boot. However, it is likely to contain keys with non-ASCII names.

Since the Unicode comparison functions available in the Windows loader do not use the Unicode uppercase table, there is no reliable way to check non-ASCII characters. The following implementation is used by the Windows loader to compare two key names (with case-insensitivity):

Fig. 11. A decompiled function used to compare Unicode strings

As you can see, for characters with codes higher than “z” (“a” + 0x19), no conversion is done. In this implementation, an uppercase version of “я” is “я”, not “Я”.

The same function implemented in the Windows kernel uses the Unicode uppercase table, so an uppercase version of “я” is “Я”. And elements in subkeys lists within the SYSTEM hive will be sorted based on this implementation. But, during the boot, they are checked using the implementation found in the Windows loader.

So, the Windows loader will consider a properly sorted subkeys list as corrupted. Then, it will delete registry keys with “offending” names.

To reproduce the issue, create two registry keys with names “*я1*” and “*Я2*” in the SYSTEM hive, under the same parent registry key, then reboot the machine. After the boot, one key, “*Я2*”, will be deleted. It happens because the subkeys list is sorted by the Windows kernel like this:

1. *я1*
2. *Я2*

In the Windows loader, this order is treated as broken:

Upcase(“*я1*”) > Upcase(“*Я2*”), because Upcase(“*я1*”) = “я1”

Microsoft confirmed the corruption but stated that it is beyond the scope, thus they won’t keep me updated.

Original vulnerability report

```
# Summary
```

A malicious usermode program can modify the ELAM hive ("C:\Windows\System32\config\ELAM"), so its blobs (registry values called "Measured", "Policy", and "Config") are correctly measured on the next boot, but the ELAM driver won't see them because registry keys containing these blobs are deleted by the NT kernel (even before the BOOT_DRIVER_CALLBACK_FUNCTION callback is registered). This results in proper (expected) PCR values but registry values (the ones previously measured) are absent when the ELAM driver tries to read them. So, the system will boot without proper ELAM signatures and this won't affect the measured boot.

Description

Steps to reproduce

(Screenshots attached.)

1. Mount the ELAM hive using a registry editor.

2. Add the "zz" key under the root key of the ELAM hive, don't assign any values to this key (see: "01-regedit.png").

3. Unmount the ELAM hive.

4. Open the ELAM hive file in a HEX editor (you can open it because it's not loaded), locate the subkeys list (subkeys of the root key), move the "zz" key to the first position on that list. (A key that was the first one before the move should now occupy the second position on the list. If there are three subkeys, just exchange the first and last keys on the list.)

The idea is to break the lexicographical order of subkeys. So, "1 2" becomes "2 1" and "1 2 3" becomes "3 2 1" (see: "02-hexeditor-intact.png" and "03-hexeditor-modified.png" for "before" and "after" states of the hive file respectively).

5. Reboot the system.

6. When the Windows loader (winload.exe or winload.efi) reads the ELAM hive, it doesn't check the lexicographical order of subkeys (see: "04-leaf-as-loaded-by-winload.png"). It's okay for the Windows loader if subkeys are stored in a wrong order.

7. When the Windows loader measures the ELAM hive (in the OslpMeasureEarlyLaunchHive routine), it reads subkeys one-by-one and measures their values (called "Measured", "Policy", and "Config").

8. If you manage to break the lexicographical order of subkeys by inserting empty keys and keeping real (non-empty) keys in the same order (relative to each other, not counting the empty keys), then the Windows loader will measure the usual (expected) data. This can be easily demonstrated with one key - "Windows Defender". If you insert the "zz" key using a registry editor, it goes to the end of the subkeys list. Like this:

- Windows Defender
- zz

If you move the "zz" key to the top (using a HEX editor), the lexicographical order is broken, but since the "zz" key has no values, it doesn't get measured. And the "Windows Defender" is measured as usual.

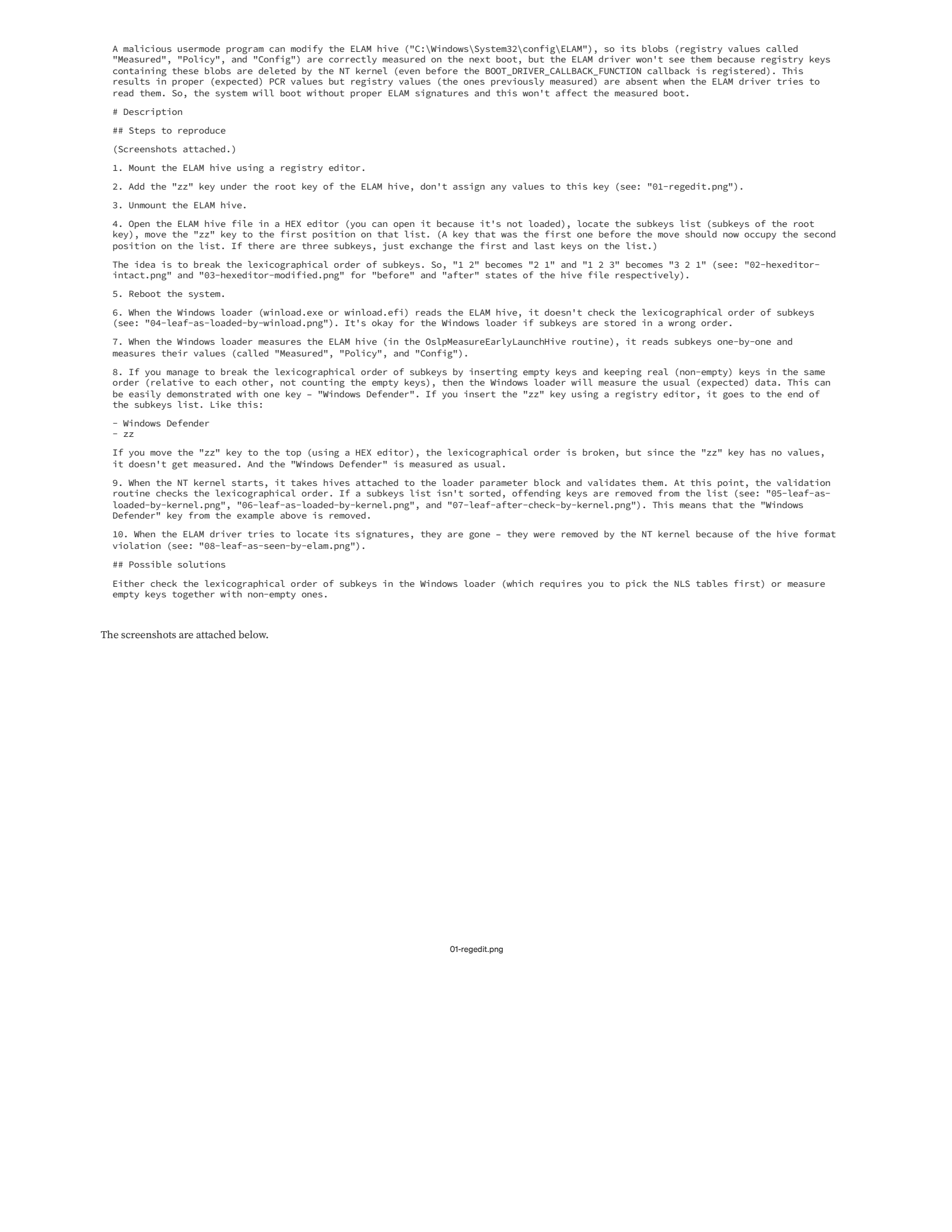
9. When the NT kernel starts, it takes hives attached to the loader parameter block and validates them. At this point, the validation routine checks the lexicographical order. If a subkeys list isn't sorted, offending keys are removed from the list (see: "05-leaf-as-loaded-by-kernel.png", "06-leaf-as-loaded-by-kernel.png", and "07-leaf-after-check-by-kernel.png"). This means that the "Windows Defender" key from the example above is removed.

10. When the ELAM driver tries to locate its signatures, they are gone - they were removed by the NT kernel because of the hive format violation (see: "08-leaf-as-seen-by-elam.png").

Possible solutions

Either check the lexicographical order of subkeys in the Windows loader (which requires you to pick the NLS tables first) or measure empty keys together with non-empty ones.

The screenshots are attached below.



02-hexeditor-intact.png

03-hexeditor-modified.png

04-leaf-as-loaded-by-winload.png

05-leaf-as-loaded-by-kernel.png

06-leaf-as-loaded-by-kernel.png

07-leaf-after-check-by-kernel.png

08-leaf-as-seen-by-elam.png

References

1. URL: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/overview
2. URL: https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process
3. URL: http://thinkinghard.com/secureinternetbanking/index.html
4. URL: https://dfir.ru/2018/07/21/a-live-forensic-distribution-executing-malicious-code-from-a-suspect-drive/
5. URL: https://source.android.com/security/verifiedboot/verified-boot
6. URL: https://dfir.ru/2018/10/07/hiding-data-in-the-registry/
7. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/early-launch-antimalware
8. URL: https://docs.microsoft.com/en-us/windows/win32/services/protecting-anti-malware-services-
9. URL: https://www.microsoft.com/security/blog/2020/09/01/force-firmware-code-to-be-measured-and-attested-by-secure-launch-on-windows-10/
10. URL: https://www.welivesecurity.com/2012/12/27/win32gapz-new-bootkit-technique/
11. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements#malware-signatures

12. URL: https://github.com/msuhanov/regf/blob/master/Windows%20registry%20file%20format%20specification.md

Appendix I. Reply from Sophos

From: Sophos.
Date: 2021-02-27. My report: 2021-02-04.

Hi,

Thanks for the interesting report. Driver issues are obviously very important and I was happy to investigate this. I've talked with our ELAM driver team and an internal security team. We don't think this is security concern, but this is not a final decision, if you disagree with the following reasoning, please let me know. There are two sensible reasons why we're marking files as KnownGoodImage.

Firstly, these files have been checked / scanned. Systems using our ELAM driver should have Sophos endpoint protection installed, which includes on-access scanning of all driver files when they're written to disk. If they were found to be malicious, we would prevent the driver installation (probably by deleting the file). Since driver files can't be active until the next boot, any driver files our ELAM driver sees have been judged clean by the endpoint protection component. They are "known good" to the best of our knowledge, so marking them as such seems appropriate.

Secondly, if a system is configured with the GPO set to only allow files marked KnownGoodImage, when legitimate drivers are updated there is a significant chance they change state to Unknown, and this can trigger a BSOD. We have had multiple customers report this on real machines. There is what I believe is a reference to this in the ELAM documentation, see section "Boot Failures":

https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements

Is that description clear? Can you see any weaknesses in the approach, or describe a scenario where an attacker could cause harm due to our behaviour?

Regarding disclosure of problems, we greatly prefer coordinated disclosure, so that if a problem is identified, a fix is available from us before disclosure occurs. Currently, I don't think a problem has been identified. We take driver level issues very seriously and if we can find a valid attack, we will want to fix this with high priority.

Thanks again for the report

Boot Security Measured Boot Windows Loader Winload Windows Kernel