The 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
21-23 September, 2017, Bucharest, Romania

Software-design for Internal Security Checks with Dynamic Integrity Measurement (DIM)

Prof. Dr. K.-O. Detken[1], M. Jahnke[1],T. Rix[1], A. Rein[2]
[1] DECOIT GmbH, Fahrenheitstraße 9, D-28359 Bremen
detken/jahnke/rix@decoit.de, http://www.decoit.de
[2] Huawei Technologies Duesseldorf GmbH, Europlatz 5, D-64293 Darmstadt,
andre.rein@huawei.com, http://www.huawei.com

Most security software tools try to detect malicious components by cryptographic hashes, signatures or based on their behavior. The former, is a widely adopted approach based on Integrity Measurement Architecture (IMA) enabling appraisal and attestation of system components. The latter, however, may induce a very long time until misbehavior of a component leads to a successful detection. Another approach is a Dynamic Runtime Attestation (DRA) based on the comparison of binary code loaded in the memory and well-known references. Since DRA is a complex approach, involving multiple related components and often complex attestation strategies, a flexible and extensible architecture is needed. In a cooperation project an architecture was designed and a Proof of Concept (PoC) successfully developed and evaluated. To achieve needed flexibility and extensibility, the implementation facilitates central components providing attestation strategies (guidelines). These guidelines define and implement the necessary steps for all relevant attestation operations, i.e. measurement, reference generation and verification.

Keywords: system state verification, memory measurment, Linux, kernel architecture, The Concise Binary Object Representation (CBOR), IMA, TCG, Trusted Platform Module (TPM).

I. INTRODUCTION

Today's systems are vulnerable to a vast majority of different attacks. In particular, attacking systems during runtime becomes more relevant each day. Very stealthy attacks target the modification of programs directly in memory after their execution; thus, common static integrity measurement cannot detect these modifications As a result Dynamic Runtime Integrity Verification and Evaluation (DRIVE) [6] was proposed. DRIVE enables detection of many memory related modifications, but often requires complex strategies for a successful attestation. As a result, it is necessary to design a flexible and extensible architecture that combines software and hardware security technologies and which supports DRIVE's attestation scheme.

In this work, we propose an architecture that implements a concept of so-called guidelines. Guidelines represent these flexible and extensible strategies for all relevant attestation steps, i.e. measurement, reference value generation and verification. In particular, our architecture and guideline implementation supports: (1) Flexibility, enabling independent system component attestation, such as user-space and kernel-space components; (2) high decoupling, by introducing triplets of functional-related guidelines that do not interfere with other triplets; and (3) extensibility, by defining clear and concise interfaces in all relevant components that facilitate a uniform intra-component communication.

To verify the feasibility of our proposed architecture, all components and various guideline-triplets for different system components were implemented, including user-space processes, Loadable Kernel Modules (LKM) and the Linux Kernel. In this work, the overall architecture and concept will be introduced and a detailed description of one particular guideline triplet will be provided, implementing a full attestation of user-space processes.

II. BACKGROUND

A. Trusted Platform Module (TPM)

A Trusted Platform Module (TPM) is a computer chip (microcontroller) that can securely store different artifacts, for instance, passwords, certificates, or encryption keys, for very different security relevant use-cases. In addition to that, another core functionality of the TPM is storing specific platform configuration measurements that can be used to securely report the configuration to an external system. Based on this reported information, an external system can determine whether the reporting system is currently in a trustworthy state or not. This reporting and verification process is called an attestation process. The TPM itself fulfills two important requirements in this concept: (1) providing authentication of reports, proofing the report originated from a particular well-known system; and (2) providing tamper-resistant reports, since the reported information is directly derived from the secure storage of the TPM providing hardware-based cryptography [2].

These designated storage areas inside the TPM are called Platform Configuration Registers (PCRs) and represent a configuration, i.e. a platform state, in form of cryptographic hash values [3], which are managed by a

process called extending, implemented by the *tpm_extend()* operation.

In turn, the *tpm_quote()* operation implements the authenticated secure reporting capability by signing a set of multiple PCRs based on a well-known TPM-key, eventually enabling an external system to verify the origin of the report.

B. Memory Management

The system memory is organized in two distinct regions: kernel space and user space memory. The kernel memory, reserved during early boot, is exclusively used for kernel relevant operations and data-structures and the user space memory is delegated to processes.

Different memory control structures and operations exist inside the kernel. These manage and control the access to different memory locations. Besides the physical memory address pointing to a concrete physical position in the system's memory, there are three types of addresses relevant in this work: (1) kernel logical addresses, (2) kernel virtual addresses and (3) user virtual addresses. The major difference is that logical addresses (LA) directly map to physical addresses (PA), whereas virtual addresses (VA) use a specific mapping mechanism and may point to memory locations that are not yet present in memory or swapped out; thus, they are not directly accessible. User VA start at address *0x0*, describe a continuous area, are isolated from one another and their layout is the same for every process. The kernel maps these VA to the actual location in physical memory, which may or may not be consecutive. Besides these, there is the bus address type, but since it is not of interest for this project, this type will not be further explained.

The kernel divides the system memory into pages with a fixed size, i.e. usually 4096 bytes on x86 and x86_64 architectures. As a result, a page is the smallest unit the kernel can manage. Each page is identified by a page table entry (PTE). A PTE contains a set of platform dependent flags that describe different properties, for instance, page state, page access rights and other metadata.

The PTE are organized in a multilevel dictionary, i.e. the page table, allowing quick lookups between a VA and its related PTE. For this purpose a so called page frame number (PFN) in encoded as part of the PTE. The PFN identifies the related page and is retrieved by performing a bit shifting operation on the PTE.

As mentioned previously, memory pages may not yet be loaded in system memory or swapped out if the system runs low of memory. When accessed, these missing pages first need to be read from the hard drive and copied into memory, which in turn is a very slow operation. However, this is only relevant for user space memory, since kernel space related pages are always present in memory. Still, whether a page is present in memory or not is indicated by the *present flag* inside the related PTE. This information is important, since the state of the other flags is undefined if a page is not present in memory.

To facilitate a simple and fast management of processes, the kernel uses several nested data structures, which are shown in figure 1. Top-level structure is the *task structure*, which represents a single process inside the kernel. The *memory management structure* (mm-structure) is nested inside task structure. It contains information about the memory that was allocated for this specific process.

Process memory is divided into segments. Each segment is represented by a *virtual memory area* (VM A) *structure*, described by user VA specifying the beginning (*VM start*) and end (*VM end*) of the segment. Additional flags, derived from the correspondent PTE flags, describe the access permission of each segment in a platform independent way. This can be any combination of: (r)ead, (w)rite or e(x)ecutable. Still, all pages that belong to a particular segment must reproduce these flags on their individual PTE. Thus, each page within a segment must have the exact same flags.

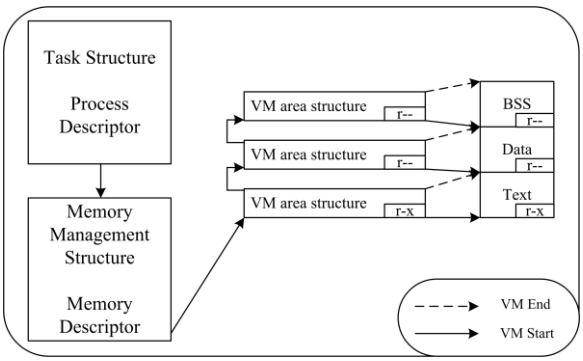

Figure 1. Linux kernel memory management

C. Integrity Measurement Architecture (IMA)

IMA is a kernel integrity subsystem to detect accidentally or maliciously altered software loaded by the OS. For this purpose, IMA maintains a runtime measurement list (RML) that is represented by a PCR inside the TPM. This anchored value represents an aggregated integrity value over the entire list with a similar construction mechanism as used by the *tpm_extend()* function. This enables the secure reporting of the anchored measurement list, since it cannot be compromised from software due to its tamper-resistant properties assured by the TPM.

Figure 2 shows the basic steps of IMA procedure. At first, IMA measures the loaded software component on the *observed system* triggered by the *mmap()* system call. Then, it generates a cryptographic hash of the file and stores it in RML. Afterwards, IMA constructs the aggregated integrity value and anchors it within TPM by facilitating the *tpm_extend()* operation. In turn, TPM computes its relevant cryptographic hash as described and stores it into a PCR. RML can then be analyzed locally or remotely by a *verifying system* in which measured values are compared to well-known references. As a result, the

verification system can detect whether a loaded software component was compromised and, in consequence, decide if the *observed system* is in a trustworthy state, i.e. behaving as intended.

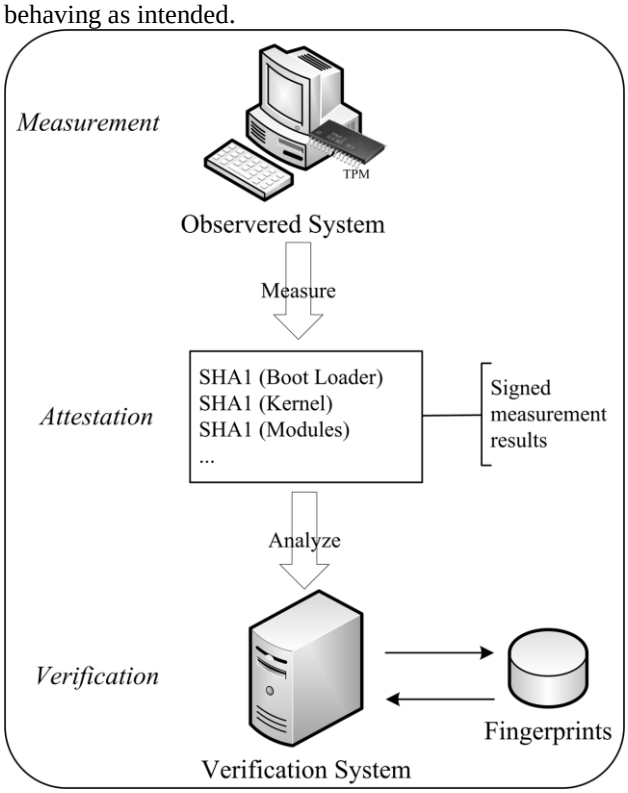

Figure 2. IMA procedure overview

III. ARCHITECTURE

In this section the designed measurement and verification system is described by outlining its structure, all components as well as their tasks and interaction.

A. High Level Architecture

From a high level point of view the whole measurement and verification system consists of three subsystems, depicted in figure 3, which will be described individually in the following sections.

On the *observed system* the *Drive Kernel Module* (DKM) represents the central main component. The DKM is responsible for performing the measurement of user space processes, loadable kernel modules and the running kernel image, by reading their actual memory contents.

The second component, R*eference Value Generator* (RVG) should ideally be run on an initially trusted system. The RVG generates reference values on the basis of well-known and trusted files, which are used to verify the results produced by the DKM. Generated reference values are stored in the *Reference Value Storage* (RVS).

To compare DKM measurement results to reference values, the *Drive Verification Component* (DVC) is required. It receives results from the DKM, searches for corresponding reference values inside RVS and compares these values. Furthermore, DVC generates a report stating the verification process result which is stored in the *verification storage* for further use.

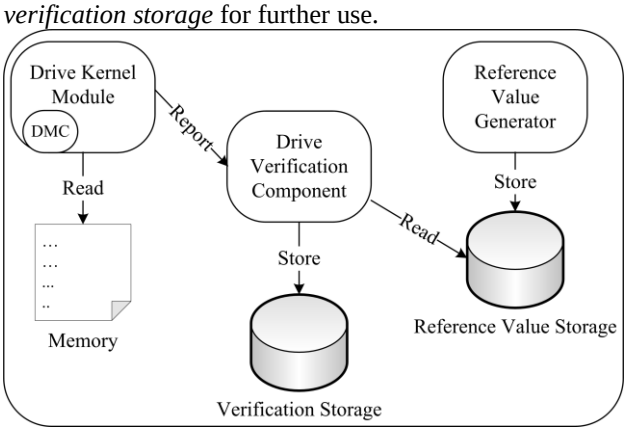

Figure 3. High-level architecture

B. Drive Kernel Module (DKM)

The DKM is a *loadable kernel module* (LKM) for the Linux kernel. It provides the framework for measurement of user and kernel space memory areas as well as access to the stored measurement results. The DKM architecture can be split into four major components.

The first component manages the communication with user space via a character device or a *SecurityFS* node, both referred to as *Drive Control Interface* (DCI) in the following. The DCI allows invoking measurements of individual targets or the whole system via a command string. For instance, a measurement target can be a single user space process, an LKM or the image of the running kernel. Additionally, a read operation on DCI is used to retrieve the list of stored measurement results. The results are encoded into a binary format and sent to the process reading from the DCI. The PoC implementation uses the *Concise Binary Object Representation* (CBOR) [4] to represent the measurement results, but the implementation can be modified to produce other binary formats as well, for instance ASN.1.

The *Drive Measurement Component* (DMC) performs the actual measurement of the target. This includes enrichment of internal data structures with additional information about the measurement target, selection and execution of matching guidelines (see section III.E Guidelines), and collection of the generated results. The measurement operations inside this area are performed asynchronously by using a Linux kernel work queue. They may either be invoked by commands received via DCI or, if configured, by a reoccurring timer. The latter is realized by injecting a delayed work package into the work queue that invokes a *Full System Measurement* (FSM) once its timer has run out. After invocation of the FSM, the work package re-queues itself into the work queue, using the same delay as before. The timer can be configured or disabled at module load time via a module parameter.

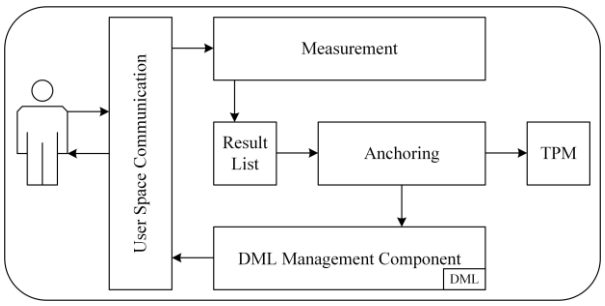

Figure 4. DKM Architecture

The measurement results from DMC are injected into a linked list, i.e. the *Result List*, storing the values for further processing. The contents of this list are consumed by the A*nchoring Component*. It processes the results generated in the previous phase by removing unnecessary information that is no longer required for the following steps. Consequently, this process generates so-called reduced results, which only contain the list of relevant measurement result information. Irrelevant data, such as information about the measurement target and timestamps created during the measurement process are dropped into save memory. The reduced results are then processed, added to the *Dynamic Measurement List* (DML) and anchored inside a trust anchor. As previously explained, the anchoring provides a secure way to prove and verify the integrity of the results during verification (see section III.D Verification).

The PoC implementation of the DKM uses a TPM chip as trust anchor. Both TPM 1.2 and TPM 2.0 are supported [5], however, the current TPM 2.0 API implementation inside the Linux kernel does only support TPM 1.2 operations. Consequently, TPM 2.0 specific features are not supported. For example, only SHA-1 hashes are supported for the PCR extend operation. On a technical level, reduced results generated in the previous step are serialized to the same binary format used for the DML output. The resulting byte string acts as input for the described *tpm_extend()* operation during the anchoring process. The to-be-extended PCR is being specified via a separate module parameter while the module is loading.

Finally, DML management is conducted by comprising all results that have been produced since the module was loaded. The responsible component manages insertion of new results into the list and allows read access to the list's content. Except for inserting new results, DML needs to be immutable.

C. Reference Value Generator (RVG)

The *Reference Value Generator* (RVG) is a complement to the DKM. It provides a framework to generate reference values that will be used to verify the integrity of the results produced by DKM. Although different guideline implementations are required for RVG, it utilizes a guideline system, which is similar to the one used by DKM (see section III.E Guidelines). The RVG itself is a user space process that should ideally be operated on a machine other than the *observed system*. The RVG requires trusted ELF files of all measurement targets meant to be measured on the DKM side. As expected, these files must match the exact same version of the files on the target system, but should originate from a secured and initially trusted reference system, guaranteeing not to be compromised at time of reference value generation.

The ELF files can be placed in a separate directory or a directory structure, for instance a copy or dump of the secure reference system. If the RVG is pointed to this directory, it will read all ELF files found and generate corresponding reference values. The exact generation process is defined by the actual guidelines implementation.

In addition to the generation of reference values, the RVG provides an interface to specify additional parameters of target systems. This includes, but is not limited to, the index of the used TPM PCR, the version of the QUOTE_INFO data structure generated by the TPM and the configuration of the running kernel. Most of these values are required to verify integrity of received DML contents.

After their generation, the reference values and the additional device parameters are stored inside RVS, which usually is a database. The RVS must be accessible by RVG during verification, explained in the following.

D. Verification

The *Drive Verification Component* (DVC) completes the set of components required for the overall system architecture. DMC verifies a set of results received from DKM against the reference values generated by RVG. The results of this verification determine whether a system is considered trustworthy or compromised. Any possible follow up actions that happen after the system state was determined, especially in cases where the system is considered compromised, is not in the scope of this component. A remote attestation system may integrate the DVC into a particular workflow, processing the verification results and take necessary remediation actions.

The input required for DVC is encapsulated in a separate data-structure, i.e. the *System State Report* (SSR). The SSR is a CBOR encoded data-structure and consists of two elements: (1) a TPM quote at time of SSR generation and (2) the DML, comprising the current measurement results. A user space tool called SSR Generator was implemented to build the SSR. It uses the TPM tools to get the current TPM quote and reads the encoded DML via the DCI.

The architecture of DVC is divided into two sub-modules, which verify different aspects of the received SSR. The first module verifies the integrity of DML by replicating the *tpm_extend()*operation and comparing the

computed result with the comprised TPM quote. This verification requires the initial PCR starting value. It then takes one result at a time from DML, performs the replicated TPM operation, and compares the result with the quote received from SSR. If they don't match, the process is repeated with the next result until no results are left or a matching result was found. If a matching result was found before reaching the end of DML, the remaining results are dropped and not verified. This is necessary because the TPM extend operation runs asynchronously which is why DML may contain results that were not anchored in time. If the end of DML was reached without a match, verification process is aborted and a corresponding failure result is written to verification storage. In this case, the received SSR was either modified during transit, for instance due to a man-in-the-middle attack or the *observed system* was modified unintendedly, since DML measurement and verification content is not coherent.

If the integrity of DML was verified successfully, DML content is processed by the second verification module. This module finally verifies the actual measurement results against the reference values generated by RVG. To find correct reference values in the reference values storage, the verification requires additional information about the target system, such as a unique identifier (e.g. the IP address), the system's patch level or other metadata.

It then processes the measurement results found in the DML, retrieves the corresponding reference values and compares them. If they do not match, the system must be considered compromised and a corresponding report is generated. The actual comparison is again done by guidelines. Each result contains information about the guideline that generated it and, consequently, the verification is aware of which guideline to call in order to verify the specific result.

Since each verification guideline is implemented in a way that exactly matches the reference values and results generated by its complement part on the DMC and RVG sides, not every result must have a matching individual reference value. It is possible for a guideline to define a result that must always match a specific value. Simply comparing the result to this value is sufficient. This means, eventually isolated cases no reference value is required.

After processing all results inside DML a report for the entire verification process is generated and stored in the verification storage. The verification can only be considered successful and valid, if the integrity verification and all individual result entries were also considered valid. If only a single entry was considered invalid, the verification state results in a failure state, which means the *observed system* is not in a trustworthy state.

E. Guidelines

The system components described in the previous sections are only frameworks that require additional business logic to perform their actual operations. This business-logic is represented as so-called guidelines. Guidelines provide a flexible and highly dynamic way to extend and modify the whole system. This is done by implementing a single interface function with code, tailored to fulfill the implementation specific behavior.

Guidelines come in sets of three implementations, one for each of the major system components. The implementations themselves are highly specialized for the actual targets they will process. Since guidelines may be added and removed from the system at any time, they must not use any functionality provided by other guidelines to prevent dependencies between them. If multiple guidelines require the exact same functionality, it should be added to the core frameworks and then referenced by the guideline implementations.

All three implementations of a guideline must agree on the results and reference values they will produce and verify. Additionally, they need to agree on a system-wide unique guideline name, which will allow the verification framework to match results with their corresponding reference values.

A DMC guideline implementation must be able to generate measurement results for at least one type of measurement target. Possible targets are user space processes, LKM or a kernel image. It must provide a public function that matches the interface definition provided by DKM. The function receives a data structure that describes the measurement target to be processed. If the guideline implementation is used for multiple target types, it must provide its own way to distinguish between those and perform the correct operations. Still, the static nature of the Linux kernel and LKM requires that the implementation of the guideline is included in DKM source code and made available to the guideline selection process. A more dynamic way to add, remove and select guidelines is planned for future developments but is not part of this PoC implementation.

RVG implementation of a guideline must generate reference values that can be processed by the verification implementation to verify results produced by DMC guideline. Current implementation of RVG requires the guideline to provide a Python module that contains a public function matching the interface definition specified by RVG. The function receives an ELF file that must be processed by guideline implementation.

Reference values themselves can be of any type. They may be simple values that are directly compared with values found in measurement results or prerequisites that allow the verification component to perform further calculations to compute actual reference values. The implementation of the particular guideline triplet contains details regarding which values are required and how these are generated. RVG must know a guideline

implementation in order to access it. This can be achieved by registering it using the command line API provided by RVG.

The third required guideline implementation is used by DVC. It must be able to receive measurement results from DML exported by DMC guideline and compare those to the reference values generated by RVG guideline. The interface function provides guideline implementation with both values; thus making it available for DVC. For some results it may be required that the verification guideline performs additional calculations to get the actual reference values for result comparison. For instance, this is required for results that contain memory offsets only known at runtime. In this case RVG generates base values required for calculation and DMC results contain additional meta-data entries with the relevant offsets, present at time of measurement. Consequently, the verification guideline must combine these base values and the received offsets and compare the result to the actual measurement result. The current implementation requires a verification guideline to be available in the verification source code which also needs to be accessed by the guideline selection mechanism. Similar to DKM, a more dynamic way to add and remove guidelines is planned for future developments.

IV. EXAMPLE GUIDELINE IMPLEMENTATION

This section will discuss an example of guideline implementation. It will describe all three parts of the guideline triplet and their individual operations at a more detailed level. A basic understanding of Linux kernel memory management might prove helpful to understand all details. Please refer to section II.B for an introduction to the required structures.

The first subsection will discuss DMC implementation. In the second subsection RVG implementation will be described and the third subsection will finally explain the process of measurement result validation. The example guideline will measure user space processes.

A. DMC implementation

Once a measurement target was selected, DMC will provide a data structure containing a reference to the kernel structure that represents the actual target. The data structure contains all information that is required for identification and processing of the target. It is passed to the guideline implementation alongside with an empty list. The guideline is expected to fill this list with its measurement results.

The goal of this example guideline is to access the memory of a user space process, read all memory segments that contain executable code, and calculate a hash digest over the contents of each segment. The hash is one of the results produced by this guideline. Additionally, it reads the access rights on the segment and tests if there are page table entries (PTE) that have a different set of flags. Both are also produced as results. This means, the guideline will return three results for each memory segment that contains executable code.

To identify and access the memory segments in question, the guideline must access the mm-structure via the task structure and iterate over the list of VM area structures contained inside, whereas each VM area structure represents a single memory segment used by this process. In addition to that, dynamically linked libraries are also mapped into the process' virtual memory. Hence, the list also contains all VM area structures of the linked libraries that need to be measured.

To identify the segments that must be processed the guideline analyses the flags inside the VM area structure whether they are readable and executable. Additionally, the guideline checks if the shared flag is not set. Only segments that are private to the process must be read. If the flags match the requirements, they are extracted and stored as a result entry. Since the flags are platform-independent, RVG can generate a bit mask that must be applied to the received flags to test for the correct set of flags. Thus, they need no further processing and can be added directly to the result list.

For each identified segment, the guideline then uses the start and end addresses of the VM area to calculate the number of pages this segment is composed of. The guideline must make sure that each of these pages is loaded into memory. As a result, it issues a request to access each page, and the kernel loads them eventually into memory. Once a page is loaded and available, its contents are read and fed into a hash algorithm[1].

Furthermore, the guideline resolves the PTE that belongs to the current page by using the mm-structure of the process and the start address of the page, which is a user virtual address. The kernel provides functions to perform this lookup. The resolved PTE is then tested for the correct flags. If the PTE flags do not match the set of flags required by the guideline, a result counter is increased.

Once all pages for a segment are processed, the guideline finalizes the hash operation, receives the digest and adds it as a result entry to the result list. Finally, the counter mentioned above is read and its value is added to the result list as well. This concludes the measurement operation on the DMC side.

B. RVG implementation

RVG operates on ELF files of the measurement target and on all libraries that are dynamically linked into a process. The process' ELF file header information shows which libraries are required.

To generate reference values for the VM area flags the guideline builds a bit mask that matches the one expected on the actual flags. This can be done because the VM area flags are a platform-independent

[1] The algorithm can be configured at module compile time default is set to SHA-1, but any algorithm supported by the kernel API may be specified.

representation of the actual PTE flags. In case of the memory segments, which are to be processed by this guideline triplet, the reference value results in 0x5. That means only read and execute flags are allowed to be set.

Generation of the reference hash digests for the memory segment contents is more complicated. First, the segment in question is located inside the ELF file and copied into a buffer. In order to do this, the ELF file is loaded with corresponding library functions and a data structure is generated that allows access to individual parts and information stored inside the ELF file.

Once the memory segment is copied, it must be page aligned. This means that it must be appended with 0-bytes until its length is a multiple of the page size of the target system. As a result, RVG allocates additional memory for the buffer and fills the allocated space with 0x0.

Once the memory segment is page aligned, the guideline calculates several different hash-digests of its contents and stores them as reference values. The values are identified by a string, representing the hash algorithm used. Hence, the verification can later identify which reference value it has to select for the comparison.

No reference value is generated for the PTE flags test, because the guideline implementations agreed on the fact that this value always must be zero to be considered valid. Any other value might indicate a compromised system.

C. *Verification implementation*

The guideline implementation on the verification side takes the measurement results provided by DMC and compares them to the reference values provided by RVG. The process of verification is simple in this case, because it requires no complex calculations during the verification process.

For each measurement target that was processed on the DMC side, the verification receives a set of result entries. This set contains all results generated for a user space process as described in section IV.A. The verification guideline iterates over all result entries and performs the necessary validation steps. For each entry, a report stating the verification state is generated and stored in the verification storage. These reports can later be used by reporting tools to generate a detailed verification report for the target system.

The PTE flags test result is a comparison to a zero value, since the guideline triplet agreed on this particular result value. If it is different from zero, this individual result entry is marked as invalid.

For the other result entries, the guideline retrieves the relevant reference values from RVS and performs further comparisons. This means, the result and reference hash digests are compared bitwise. Only if both values are equal, the result is considered valid.

To test the VM area flags, the set of flags received from the DMC is compared to the corresponding reference value. This is done by using bit-wise operations. First, a bit-wise AND is calculated on the basis of the result and a bit-mask that has all four relevant bits set. These are the read, write, execute, and shared flags, which match a mask of 0xF. Afterwards, the result of this operation is compared to the reference value. If they are equal, the required flags are set accordingly and all other flags are not. In this case the result is considered valid. Otherwise, it is considered invalid.

In case no reference value can be found for a specific result entry, the entry is also considered invalid. This is because unexpected elements must have been found on the target system, which are not meant to be present for the process. The same is true for reference values without a matching measurement entry. In this case an expected segment is missing on the target system.

DMC implementation will only provide the configured digest of the segment. This means, once one hash digest was found and verified, the other reference values for different hash digests are ignored.

The guideline execution stops once all provided result entries are processed.

V. Conclusion and Future Work

In this work, instantiated architecture for secure measurement and verification of Dynamic Runtime Information for Linux based OS was presented. Our solution facilitates hardware based security technologies and enables an attestation on the basis of introduced guidelines. The presented guidelines represent a particular triplet set of rules, to attest a targeted system, on the basis of generated reference values. The application of a triplet-set on the basis of user space processes was demonstrated and all necessary steps to conduct a secure attestation were described. In future work, further capabilities of guidelines will be exploited and concrete guideline triplets for more complicated attestation schemes, such as LKM or the Linux kernel, will be provided.

References

[1] Arun K. Kanuparthi: *Architecture Support for Dynamic Integrity Checking.* IEEE Transactions on Information Forensics and Security, Vol. 7, No. 1, February 2012. p. 321-332, DOI: 10.1109/TIFS.2011.2166960

[2] Trusted Computing Group: *Trusted Platform Module (TPM) Summary*. https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/, Date Published: April, 01, 2008

[3] Will Arthur, David Challener, *A Practical Guide to TPM 2.0*, P13, EAN: 9781430265849

[4] C. Bormann, P. Hoffman: *Concise Binary Object Representation (CBOR)*. Internet Engineering Task Force (IETF), RFC 7049, IETF in October 2013

[5] Trusted Computing Group: *Trusted Platform Module (TPM) 2.0*. a brief introduction, https://www.trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf, 2015

[6] Andre Rein: DRIVE: Dynamic Runtime Integrity Verification and Evaluation. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17), 2017, DOI: https://doi.org/10.1145/3052973.3052975