DRSA: Debug Register-Based Self-relocating Attack Against Software-Based Remote Authentication

Zheng Zhang[1], Jingfeng Xue[1], Tianshi Mu[2], Ting Yu[2], Kefan Qiu[1], Tian Chen[1], and Yuanzhang Li[1(✉)]

[1] Beijing Institute of Technology, Beijing 100081, China
popular@bit.edu.cn
[2] China Southern Power Grid Digital Grid Group Co., Ltd., Guangzhou 510000, Guangdong, China

Abstract. Remote attestation (RA) is an essential feature in many security protocols to verify the memory integrity of remote embedded (IoT) devices. Several RA techniques have been proposed to verify the remote device binary at the time when a checksum function is executed over a specific memory region. A self-relocating malware may try to move itself to avoid being "caught" by the checksum function because the attestation provides no information about the device binary before the current checksum function execution or between consecutive checksum function executions. Several software-based that lack of dedicated hardware rely on detecting the extra latency incurred by the moving process of self-relocating malware by setting tight time constraints. In this paper, we demonstrate the shortcomings of existing software-based approaches by presenting Debug Register-based Self-relocating Attack (DRSA). DRSA monitors the execution of the checksum function using the debug registers and erases itself before the next attestation. Our evaluation demonstrates that DRSA incurs low overhead, and it is extremely difficult for the verifier to detect it.

Keywords: Remote attestation · Debug registers · Self-relocating malware

1 Introduction

With the development of the Internet of Things and the increase in the variety and number of special-purpose embedded devices, the security threats faced by these devices are increasing. Preferred targets of malware change from general-purpose computers to numerous and inter-connected embedded devices mainly because of the latter's lack of security protections. As our society is gradually becoming surrounded by these low-end embedded devices, their safety cannot be ignored, although security is typically not the highest priority due to cost, power constraints, or size.

J. Chen and Z. Xia (Eds.), BlockTEA 2023, LNICST 577, pp. 23–40, 2024.
https://doi.org/10.1007/978-3-031-60037-1_2

For such devices that have no means to prevent malware attacks, Remote Attestation (RA) is a distinct security service for detecting malware on embedded devices. RA is typically realized as a challenge-response protocol. In this protocol, RA allows a trusted Verifier ($\mathcal{VRF}$) to compute a checksum of an untrusted Prover ($\mathcal{PRV}$)'s memory to attest its integrity. Since the $\mathcal{VRF}$ is assumed to know the hardware configuration and exact memory contents of the $\mathcal{PRV}$, it can compute the expected checksum and compare it with the received one. Mismatched values indicate that the device has most likely been compromised. Software-based RA [16–18,20] is an approach particularly suitable for verifying the trustworthiness of low-end embedded devices because it does not rely on hardware to control the execution of the integrity-ensuring function. Compared with hybrid RA (based on hardware/software co-design) [1,6,10] and hardware-based RA (e.g., those based on a TPM [11] or other dedicated hardware modules), which require some level of hardware support, software-based RA is less costly.

Most of the previously proposed software-based RA techniques perform an integrity check on the whole memory content of a low-end sensor node because the memory of this kind of device is usually small enough. Due to the rapid technological growth, the usage of middle-end and high-end devices is increasing. These devices can perform tentative computations such as deep learning network training [13,22], executing complex protocols and authentication algorithms [23, 24]. In middle-end and high-end devices that have enough resources, such as a powerful processor with x86 architecture, to run a traditional OS such as Linux, however, attesting the whole memory content is time-consuming.

In this paper, we emphasize that the security of software-based attestation on general-purpose operating systems is uncertain. We present a Debug Register-based Self-relocating Attack (DRSA), a malicious program that circumvents attestation by relocating itself and restoring the contents before it is measured. The implementation of this attack uses debug registers provided by the debugging architecture, which is available on most processors today. In general, debug registers can support software/hardware breakpoints, and the trigger conditions are various. DRSA utilizes debug registers to monitor the beginning and the end of an RA measurement, erases itself, and restores the original program memory contents before attestation. The malicious code can be restored after the RA procedure to regain control over the compromised device. We evaluate DRSA and show the shortcomings of software-based attestation design.

The remainder of this paper is organized as follows: Sect. 2 provides a preliminary on software-based attestation technologies and the x86 debugging architecture. Section 3 discusses the threat model and assumptions of this work. Section 4 presents a detailed description of DRSA. Section 5 introduces the design and implementation details. The effectiveness and efficiency of DRSA are evaluated and discussed in Sect. 6. Section 7 discusses related work and Sect. 8 concludes this paper.

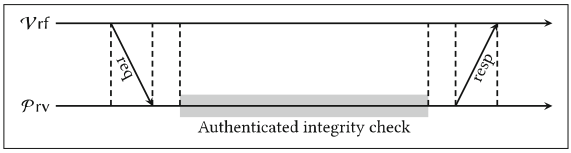

Fig. 1. Timeline of a typical RA protocol

2 Background

2.1 Software-Based Attestation

Existing software-based RA techniques are all realized as a challenge-response protocol where the $\mathcal{VRF}$ challenges a $\mathcal{PRV}$ (a target untrusted remote device) to compute a checksum of its specified memory region. We first describe the basic challenge-response protocol and then list the existing software-based RA schemes.

Challenge-Response Protocol. A typical challenge-response paradigm uses a checksum function to compute the checksum of the memory content, the procedure of which can be described as follows:

(1) $\mathcal{VRF}$ sends an attestation request with a challenge to $\mathcal{PRV}$. This request also contains a nonce generated by the $\mathcal{VRF}$ to be involved in the checksum computing to prevent pre-computation or replay attacks.
(2) $\mathcal{PRV}$ receives the request and computes a challenge-based authenticated integrity check over a pre-defined memory region. Depending on the attestation purpose, this region corresponds to either the entire program memory to ensure that no malware exists or only a specific memory region to obtain the guarantee of a dynamic root on the untrusted platform [16].
(3) $\mathcal{PRV}$ returns the result to $\mathcal{VRF}$.
(4) $\mathcal{VRF}$ receives the result and checks whether it matches the expected checksum.

The timeline illustrating this attestation process is shown in Fig. 1. Since the usual RA threat model assumes that the target device is fully compromised, software-based RA must protect the integrity-ensuring function from the potential malware without any hardware support. Prior works rely on strict time constraints or filling the empty memory locations to prevent malicious code.

Time Constraint. A compromised $\mathcal{PRV}$ may try to evade the detection of software-based RA schemes by performing a memory copy attack. This attack modifies the original code and redirects the memory accesses to a correct copy of the original code that is stored elsewhere in memory. Some software-based RA relies on a strict time constraint and optimized attestation code to defend against memory copy attack because they claim that the overhead caused by

redirection would be easily detected by the $\mathcal{VRF}$. Checksum functions of these approaches must, therefore, be simple and highly time-optimized, leaving no room for the adversary to compress the attest time. However, it is difficult to assess what the best attack against a certain approach is, which makes the security of this approach uncertain. Several attacks have been proposed [7] and demonstrated the weaknesses of time constraints. Moreover, most time-based attestation require disabling interrupts during attestation procedure execution to ensure that the adversary can not move malware around during attestation by forcing an interrupt. This will affect the normal operation of the device and result in low availability.

Memory Filling. To overcome the weakness of tight time constraints, some software-based RA schemes fill the free program memory with random noise before deployment so that there is no empty space for the adversary to store its malware. These approaches claim that the attacker can not simply overwrite these noises because they are involved in the calculation of the checksum. However, this method still has disadvantages. In fact, the adversary can compress the original code to gain free space for storing its malware and executing it. Moreover, memory filling is only suitable for sensor devices where the memory layout is known in advance. For some implementations based on complex instruction sets and operating systems, the situation will be more complicated. For example, Pioneer is a software-based RA implemented on x86 architecture with Linux kernel, the memory layout of which may change dynamically during program execution due to the dynamic memory allocation. Memory filling will not work because the pre-filled noises may be overwritten during program execution.

2.2 X86 Debugging Architecture

The debugging architecture is supported by most of the processors today to facilitate on-chip debugging. There are a total of eight debug registers (DR0 through DR7) in x86 architecture to support both debug exceptions and hardware breakpoints required by software debuggers [12]. Among them, DR0 to DR3 are used to specify memory addresses or I/O locations to be monitored by the debugger. The processor raises a debug exception when an instruction or memory address matches the address that is stored in one of these four registers. The address monitoring process is performed in parallel with the normal virtual to physical address translation, and thus no additional expensive intercept-and-check in software is required. DR4 and DR5 are reserved, while DR6 is used to report the status of the debugging conditions when a debugging exception is raised. Bn bit of DR6 indicates that the nth breakpoint was reached. BS and BT bits signify the exception is triggered by single stepping and task switching, respectively. DR7 is used to control/configure the trigger condition of the debug exception when the address matches. By setting the corresponding Ln or Gn bit in DR7, the nth breakpoint could be enabled or disabled. Ln bit and Gn bit enable the

nth breakpoint for the current task and all tasks, respectively. The corresponding breakpoint is disabled when both Ln and Gn bits are cleared. R/Wn field configures the trigger mode of the nth breakpoint. For example, the data read or write in the monitored memory address will raise a debugging exception if value 11 is set. The instruction execution of the specific memory address can also be realized by monitoring the set value 00. The size of the monitored memory location of the nth breakpoint can be set through the LENn field.

3 Threat Model and Assumptions

3.1 Hardware Platform Description

DRSA is implemented on devices with x86 architecture. Unlike the Harvard memory architecture, the program and data memories of x86 architecture devices are not physically separated. This makes self-relocating malware more difficult to evade attestation on x86 architecture. In fact, the separation of program memory and data memory gives the attacker a vulnerability to evade attestation. The attacker can hide malicious code in data memory and pass the attestation protocol that does not attest data memory. While the presented attack is validated on an x86 architecture-based device, it is not specific to x86 architecture. It exploits the characteristics of the debugging architecture supported by the processor. The proposed attack is applicable to any device that uses similar debugging features, such as tracing and monitoring.

3.2 Adversarial Assumptions

In this work, we assume that the adversary aims to install its malicious code in the memory of the target device and avoid being detected during the attestation process. The adversary has complete control over the code and data of the target device before and after attestation. We also assume that the attacker has no direct control of the device at attestation time. The attack succeeds if the device passes the attestation while the malicious code resides in memory. We do not address the details of how attackers install malicious code on devices because it is beyond the scope of this paper. Finally, we assume that the attacker does not perform hardware attacks on the target device. Specifically, it does not modify the device hardware or induce hardware faults.

4 Debug Register-Based Self-relocating Attack

In this section, we introduce the implementation of DRSA. The attack is formed based on the debugging features of x86 debugging architecture. It circumvents malware detection by detecting the start of the attestation and restoring the original contents before it is measured. This is achieved by monitoring instruction execution and data access events using debug registers.

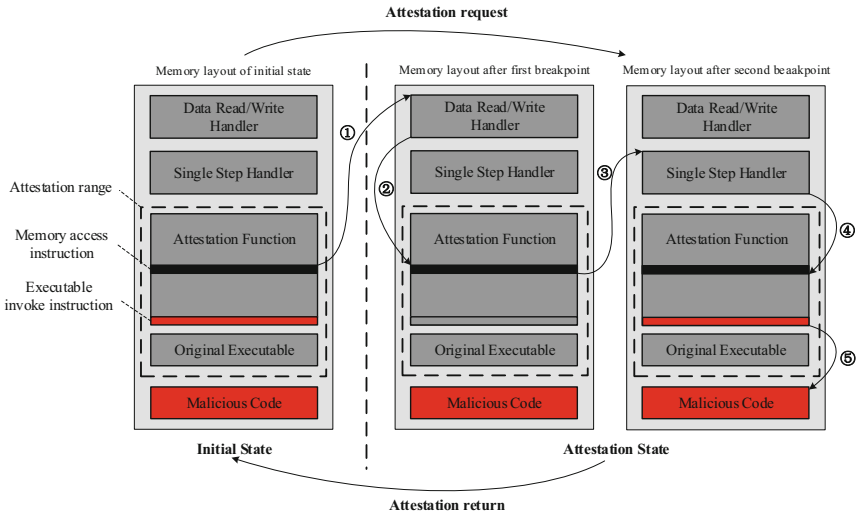

Fig. 2. Overview of DRSA. The numbers represent the transfer order of the control flow.

Figure 2 illustrates the overview of DRSA. The memory layout of the target device is also depicted. The core of the attack is composed of a data read and write handler, a single-step handler, a hook (a jump instruction), and malicious code. When an attestation request is received, the target device switches from the initial state to the attestation state, and its memory layout also changes. The numbers represent the transfer order of the control flow. As stated above, the memory layout of the target device with x86 architecture is not divided into program memory and data memory. Moreover, the existing remote attestation routine for x86 architecture, Pioneer, only verifies part of the memory content rather than the entire memory [16]. The checksum code of Pioneer is self-checksumming and computes a checksum over the entire attestation function and the executable. The correct checksum provides a guarantee that the verification function code is unmodified and further establishes a trusted computing base called the dynamic root of trust. Once established, the dynamic root of trust verifies the integrity of the executable and invokes it. The executable is guaranteed to be trusted if it is unmodified and executed uninterrupted. DRSA works by inserting a hook (a jump instruction) into the attestation routine to replace the invoke instruction during the attestation procedure function. When the dynamic root of trust tries to invoke the origin executable, the control flow will be hijacked, and malicious code will be invoked.

However, simply modifying the attestation function will be detected during the attestation procedure because it will lead to an invalid checksum. To escape detection, DRSA utilizes two types of hardware breakpoints to monitor the checksum function's operation of reading the memory content and modifying

the memory content in the corresponding exception handlers. Specifically, the following is a detailed description of the attack scenario presented in Fig. 1:

1. The malicious code is installed on the victim device beforehand, and the existing exception handlers are replaced with our own handlers to implement DRSA's core function. Next, we insert a hook to the attestation code by replacing the executable invoke instruction of the attestation function with a jump.
2. A data read/write breakpoint and a single-step breakpoint are set, respectively, to trigger the corresponding exception handler. Specifically, the data read/write breakpoint is set beforehand, while the single-step breakpoint is set at runtime.
3. When an attestation request is received, the victim device switches to the attestation state. The attestation function will access the memory of the attestation region and compute the checksum. The data read/write breakpoint will be triggered when the attestation function accesses the address of the hook, and the data read/write exception handler will be invoked.
4. The data read/write exception handler receives control and sets the original executable invoke instruction back to what it was. Then the single-step breakpoint is set before the control flow transfers back to execute the memory access instruction.
5. The single-step breakpoint is triggered after the single memory access instruction executes. The single-step exception handler receives control and installs the hook to replace the executable invoke instruction of the attestation function again. Then the control flow transfers back to continue executing the next instruction of the memory access instruction.
6. The attestation function will compute a valid checksum because the modified memory content is restored to the original content before being accessed by its memory access instruction. The victim device switches back to the initial state, but the hook, however, is still present in the memory without being detected by the verifier.

The attestation function is, therefore, executed over the clean memory of the attestation region. Since neither the exception handlers nor the malicious code are in the attestation region, their presence will not affect the checksum result. Thus, the valid checksum is sent to the verifier, and the integrity of the executable will be measured. Since the content of the executable has not been modified, it can pass the integrity measurement and is ready to be invoked. Once the original executable is invoked, the control flow will be redirected to execute the malicious code.

5 Design and Implementation

To achieve the attack process mentioned above, DRSA utilizes debugging architecture, which provides hardware breakpoints and exception mechanisms to transfer execution to custom handlers, which relocates malicious content before

being detected. To achieve this, the corresponding bits of the debug registers should be set correctly. In this section, we explain the detailed setting of the debug registers and the key implementation steps of DRSA under the x86 architecture.

5.1 Memory Restoring via Data Read and Write Breakpoint

DRSA uses the debugging architecture of the underlying platform to provide hardware breakpoints on code execution, data, and memory access. Support for debugging architecture on most processors is in the form of debug registers. The breakpoint trigger conditions can have various attributes such as execution, write, read/write, etc. To achieve memory restoration via data read and write breakpoint, the address and attributes of the breakpoint should be set through corresponding registers.

In the x86 architecture, current processors typically provide 4 locations of hardware breakpoints, the addresses of which can be set through DR0-DR3. We use the first debug register, DR0, and set its value as the address of the hook to monitor any memory access to it. The specific attributes of the breakpoint are controlled by DR7. Table 1 shows the detailed setting of each bit of DR7. L0 bit and G0 bit control the scope of the first breakpoint. It can be set as persistent or nonpersistent during its creation by setting the G0 bit to 1 or setting the L0 bit to 1, respectively. We set the G0 bit to 1 so that every access to the hook address will trigger the read and write breakpoint. GD bit is set to prevent debug registers from being modified by other programs. Instructions that access debug registers will raise an exception, and the relevant information on debugging exceptions will be recorded by DR6. R/W0 is controlled by bits 16 and 17 of DR7, which determines the trigger condition of the breakpoint. We set R/W0 to 01 so that the breakpoint will be triggered by the read/write operation of the address set by DR0. Finally, LEN0 is set to 00 to specify the memory size monitored by the breakpoint as 1 byte.

Table 1. Dr7 settings for data read/write breakpoint.

Bit Name	Bit Number	value	Description
L0	0	1	the breakpoint will only be valid for the current task
G0	1	0	the breakpoint will be valid for all the tasks
GD	13	1	instructions that access debug registers will raise an exception
R/W0	16&17	01	the breakpoint will be triggered by read or write operations
LEN0	18&19	00	specify the memory size monitored by the breakpoint as 1 byte

DRSA installs its own data read and write handler, replacing the existing handler to implement the restoration of original memory content. When the data read and write handler is invoked due to a memory access exception, the handler first erases the hook code and then restores the original memory contents. Finally,

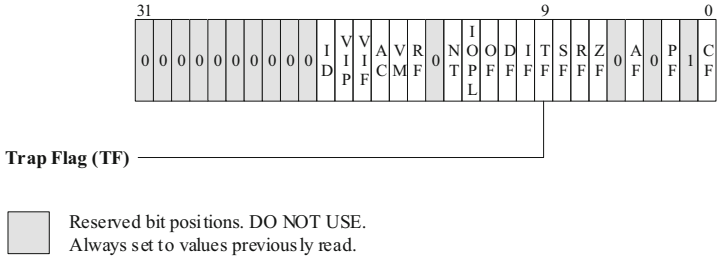

Fig. 3. The trap flag in the EFLAGS register

the handler prepares for a single-step exception and transfers the control flow back to the attestation function.

5.2 Hook Re-installing via Single-Stepping

DRSA employs a single-step exception for hook re-installing. The single-step exception is a standard exception on all processor architectures that are triggered upon execution of each instruction when a certain processor flag (also called the trap flag) is activated. The exception generated by a single step is consistent with the exception generated by the hardware breakpoint. Thus, the single-step exception makes use of the trap flag for its functioning. In the x86 architecture, the trap flag can be set through the EFLAGS register. The position of the trap flag in the EFLAGS register is shown in Fig. 3. DRSA installs its own single-step handler, replacing the existing handler to implement hook re-installing. The single-step handler is installed on demand from the data read and write handler.

In the data read and write handler, the trap flag (bit 8) is set to 1 to enable single-step mode for debugging before the control flow transfers back to execute the memory access instruction of the attestation function. Then, the single-step exception will be triggered after the execution of the memory access instruction. When the single-step handler is invoked due to a single-step exception, the handler reinstalls the hook back to replace the executable invoke instruction of the attestation function again. Then, the handler resets the trap flag, uninstalls itself, and issues an end of exception.

5.3 Framework Composition and Source Organization Under Windows

The framework of DRSA is implemented in the form of a loadable kernel mode driver under the Windows operating system that is able to monitor the memory access operation initiated by the attestation function and relocate malicious content before it is detected. Figure 4 illustrates how DRSA performs memory access monitor and malicious content relocating. The left part is the framework

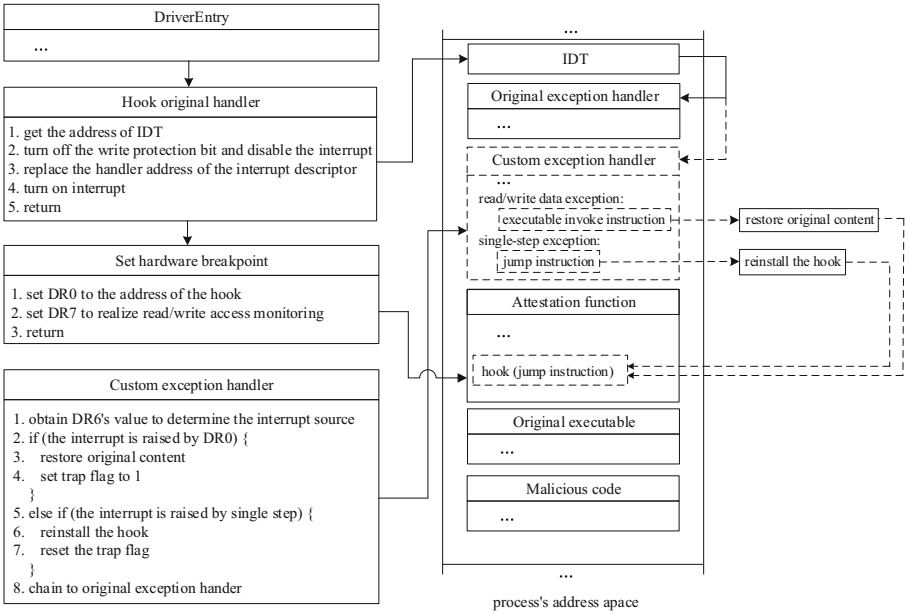

Fig. 4. DRSA in the form of a kernel mode driver

of the DRSA in the Windows driver form, while the right part is the attestation process's address space. The driver is mainly composed of three key parts (pseudo-code shown in Fig. 4):

Hook Original Handler: To install its own exception handlers, DRSA intercepts calls to the Interrupt Descriptor Table (IDT) and adds in its own processing. After the DriverEntry initializes driver-wide data structures and resources, the DRSA driver creates a hook to replace the original exception handler. Specifically, it first obtains the base address of the IDT and then extracts the address of the corresponding original handler function from an interrupt descriptor. The write protect bit is turned off (by modifying the CR0 register), and the interrupt is disabled (by using the CLI instruction) to avoid being interrupted by other interrupts when modifying the IDT. Finally, it points the target interrupt descriptor to a new handler function, reloading the CR0 register and enabling interrupts. When an exception is raised, the custom exception handler function will be invoked instead of the original exception handler.

Set Hardware Breakpoint: The data read/write breakpoint is set during the driver's loading process. It mainly includes setting the DR0 register, and the DR7 register. Specifically, the DR0 register is assigned the address of the hook we installed in the attestation function, while the DR7 register is assigned the specific value summarized in Sect. 5.1 so that the trigger condition of the breakpoint is set to read/write. During the driver development routine, we achieve direct control of the DR registers through inline assembly.

Custom Exception Handler: The custom exception handler is developed as part of the driver and loaded into memory when the driver is installed. To achieve malicious code relocating, the custom exception handler keeps a copy of the original memory content and malicious code to be installed in the attestation code. When it is invoked by a raised exception, the value of the DR6 register is first obtained to determine the interrupt source. The value of BD (bit 13), as mentioned in Sect. 5.1, indicates whether the exception is triggered by the breakpoint set by the DR0 register. The BD bit is 1, indicating that there is a program trying to access the memory address (read or write) monitored by the DR0 register. Then, the custom exception handler will restore the original content of the attestation code (an executable invoke instruction) and set the trap flag of the EFLAGS register to 1. Next, it will chain to the original exception handler to complete its original function. If the exception is not raised by the DR0 register, and the trap flag is set, which means that the exception is raised by single stepping, the custom exception handler will reinstall the hook and reset the trap flag and chain to the original exception handler. Other interrupt sources will not trigger the execution of additional code in the custom handler and will transfer the control flow to execute the original handler instead.

Owing to the capabilities that debug architecture provides, it is clear that the malicious modification of the target memory content can successfully escape the detection of the attestation function.

5.4 Framework Composition and Source Organization Under Linux

DRSA under Linux OS is implemented using ptrace, a system call provided by Unix, and several Unix-like operating systems. By attaching to another process using the ptrace call, a tool has extensive control over the operation of its target. It can also single-step through the target's code and observe and intercept system calls and their results. More importantly, ptrace provides requests to control the debug registers directly, which facilitates the implementation of our attack method. The framework of DRSA based on ptrace is shown in Fig. 5. DRSA is implemented as one single process (the "tracer") to observe and alter the execution of the attestation process (the "tracee"), achieving malicious code relocation at runtime.

The left part of Fig. 5 illustrates the execution flow of DRSA based on price. DRSA process commences tracing by having the attestation process do a $PTRACE\_ATTACH$. Then, the attestation process will stop, and the DRSA process can use various ptrace requests to inspect and modify the attestation process. DRSA process first sets data read/write breakpoints at the hook address of the attestation process. By using ptrace requests $PTRACE\_POKEUSER$, the DRSA process sets the DR0 register and DR7 register to the specific values summarized in Sect. 5.1. Then, the call $PTRACE\_CONT$ is delivered to continue the execution of the attestation process. The attestation process will stop again when the attestation function triggers the data read/write breakpoint. The DRSA process restores the original content of the attestation function by using $PTRACE\_POKETEXT$, which can modify the code segment of the tracee

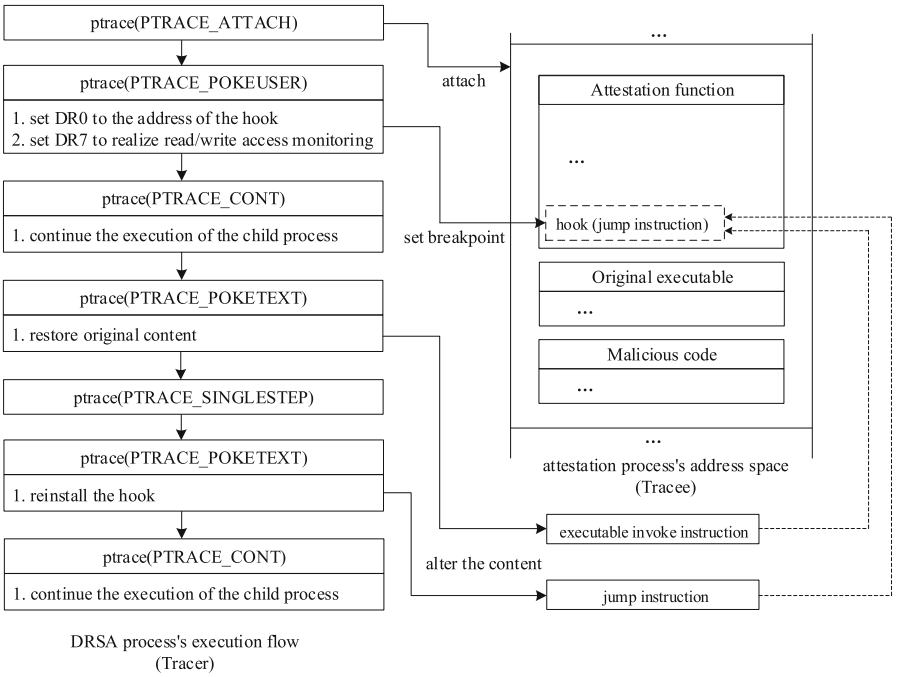

Fig. 5. Implementation of DRSA under Linux

directly. After that, the DRSA process issues a *PTRACE_SINGLESTEP* request to restart the stopped attestation process but arranges for it to be stopped after the execution of a single instruction (the memory access instruction in our scenario). The hook code (a jump instruction) will be reinstalled in the attestation function during this suspension, and the DRSA process then makes the place-stopped attestation process run again by using *PTRACE_CONT*.

By utilizing ptrace, the attestation process will not be aware of the existence of the DRSA process. Additionally, attaching to the tracee will not add additional instructions in the attestation region, which will not affect the result of the checksum calculation.

6 Evaluation

In this section, we measured the effectiveness of DRSA on some specific code attestation schemes. We tested our attack on SWATT [17] to compare with existing attack methods against the existing remote attestation routine. The untrusted platform used in our experiment is a PC with a 3.41 GHz Intel i7-6700 processor running Windows 10.

6.1 Attestation Time Overhead

In this subsection, we tested the time overhead of DRSA on existing protocols. Our goal is to show that the proposed attack could escape from the detection of the target attestation routine.

We tested DRSA on our implementation of SWATT. The original SWATT was implemented based on an ATMega163L microcontroller with 16 KBytes of program memory. In order to simulate SWATT on the x86 platform, we ported SWATT's attestation function and made it verify the specified 128 KBytes memory region, including itself. We further implemented an attacker's version that assumes that DRSA has compromised the attestation function of SWATT. Table 2 shows the results indicating the attack overhead introduced by DRSA. We compared it with the original SWATT attack. We also compared the attack overhead imposed by DRSA with those introduced by [7], including the memory shadowing attack and ROP attack. Note that we obtained the overhead of ROP attack measures from the articles previously published because it can only be implemented on a Harvard memory architecture.

Table 2. Overhead of different attacks.

Method	Time of Execution (ms)	Attack Overhead (ms)	Attack Overhead (%)
Original SWATT	11061	–	–
Original SWATT Attack	–	–	13%
SWATT (MicaZ)	13103	–	–
ROP Attack (MicaZ)	–	42.3	0.32%
SWATT (x86)	56.5	–	–
Shadow Attack (x86)	60.1	3.7	6.47%
DRSA	**56.9**	**0.4**	**0.71%**

To facilitate the comparison of different attack overhead, timings are collected on three implementation versions of SWATT. The original SWATT attack represents the best attack expected by the designers of SWATT, which checks each address generated in the pseudo-random sequence. This attack would add three cycles for test and redirection. Since the original attestation function main loop is 23 cycles long, the malicious attestation function incurs 13% attack overhead. The ROP attack is implemented on a MicaZ device with 128 KBytes of program memory in [7]. The authors ported SWATT on MicaZ and tested their ROP attack on it. The experimental results show that the time required for the ROP attack to hide the malicious code is less than 50 milliseconds, introducing about 0.32% overhead.

The last three rows in Table 2 summarize the execution time of x86-based SWATT, the attack overhead of DRSA, and the shadow attack based on our x86 platform. On the x86 platform, the number of SWATT cycles should be increased, according to the Coupon's Collector Problem. Due to the high CPU

frequency of the experimental platform, we further double the number of iterations to get more accurate results. Compared to the running time of x86-based SWATT, DRSA introduced 0.71% of overhead. The experimental results show that the attack overhead of DRSA is much smaller than the original SWATT attack and shadow attack. While it is a bit slower than the ROP attack, this delay is hard to be detected by the verifier. The main reason for the difference in the overhead of DRSA and ROP attack is that the ROP attack inserts the hook and triggers the rootkit hiding functionality, which will delete the rootkit code from the program memory at the beginning of the attestation function. This means that additional malicious code is executed only once during an entire attestation routine. In DRSA, however, the execution times of the exception handlers depend on the number of times the modified memory location is accessed during an attestation routine. Therefore, additional malicious code may be triggered multiple times, which incurs more execution time.

6.2 Status Information on the Stack

The authors of [16] have taken into account that an adversary generates a non-maskable interrupt or exception by setting a breakpoint to gain control. Their defense method is to make use of the stack to store a part of the checksum. They claimed that all CPUs automatically save some state on the stack when an interrupt or exception occurs. If the stack pointer is pointing to the checksum that is on the stack, any interrupt or exception will cause the processor to overwrite the checksum. However, this stack trick does not work for DRSA because the status information will be saved in debug registers instead of on the stack if the exception is generated by a hardware breakpoint.

To confirm this conclusion, we obtained the address of the stack pointer before and after the execution of the exception handlers to verify whether it has changed. We conducted the experiment on the ptrace-based DRSA. The experiment platform used in our experiment is a PC with a 3.41 GHz Intel i7-6700 processor running the Ubuntu 18.04 operating system. By using *PTRACE_GETREGS* request, the DRSA process (the "tracer") could copy the general-purpose or floating-point registers of the attestation process (the "tracee") to its address data. By accessing the RSP register, we can obtain the stack pointer. In order to compare the difference between hardware breakpoints and software breakpoints, we implemented a process of gaining control from the attestation function through a software breakpoint. As shown in Fig. 6, the first instruction of the attestation function is replaced with an INT3 instruction. This instruction is a one-byte instruction defined to temporarily replace an instruction in a running program in order to set a software breakpoint. When the attestation code tries to run, a debug exception will be raised, thereby gaining control. We also record the stack pointer before and after the software breakpoint is triggered. The experimental results are shown in Table 3.

The experimental results illustrate that exceptions raised by the data read/write hardware breakpoint and single stepping in DRSA do not cause the stack pointer to move. In a comparative experiment, however, the stack pointer

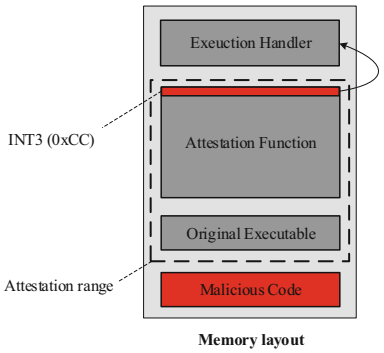

Fig. 6. Gaining control from the attestation function through a software breakpoint

Table 3. The value of the stack pointer. Stack pointer before the breakpoint triggers (sp/ Before.bp), stack pointer after the breakpoint triggers (sp/ After.bp) and stack pointer after the single stepping (sp/ After.ss).

Breakpoint Type	sp/ Before.bp	sp/ After.bp	sp/ After.ss
Software Breakpoint	$0 \times 7FFF6F6DAB78$	$0 \times 7FFF6F6DAB60$	–
Hardware Breakpoint (DRSA)	$0 \times 7FFCDE29DDB8$	$0 \times 7FFCDE29DDB8$	$0 \times 7FFCDE29DDB8$

points to different locations before and after the execution of INT3 instruction. This will cause the checksum to be overwritten and further cause the manipulations to be detected by the attestation function. We conclude that DRSA can escape from the detection of the existing attestation routine on the x86 platform.

7 Related Work

A. Software Attestation

SWATT [17] is an early software attestation scheme based on challenge-response protocol. It requires a strict running time of the attestation procedure to prevent an attacker from redirecting the memory regions. The fact that SWATT relies on optimized function execution and a strict time limit is considered a drawback. Pioneer [16] is a software attestation scheme similar to SWATT, focusing on the x86 platform. Considering that the target devices are equipped with a large amount of memory, Pioneer is self-checksumming and establishes a dynamic root of trust to invoke a trusted executable. Pioneer also requires strict time constraints to detect memory copy attacks. Distributed [21] uses memory filling to overcome the time constraints imposed by SWATT. The free memory of the device is filled with randomness before deployment, preventing the attacker from having empty memory to evade detection during attestation. However, schemes based on memory-filling techniques are threatened by compression attacks, which compress the original program of the victim device, opening up space for malicious code.

B. Hardware Attestation and Hybrid-based Attestation

To address limited security protections of software attestation schemes, hardware-based approaches rely on trusted computing architectures such as TPM [11], ARM TrustZone [4], Sancus [14]. Despite their strong security guarantees, the requirement for costly customized hardware that cannot be accommodated in small IoT platforms makes hardware-based protocols incompatible with many low-end devices. To this end, hybrid-based solutions, such as TyTAN [5], leverage the best properties of software-based and hardware-based RA approaches to establish Root-of-Trust by relying on minimal hardware assumptions. In particular, hybrid solutions require modification of device hardware to ensure atomic and secure code execution of RA protocols. One recent work that aims to fill the gap between software-based and hybrid-based RA schemes is SIMPLE [3], a hypervisor-based RA scheme for resource-constrained IoT devices. SIMPLE relies on a software-based memory isolation technique called Security MicroVisor (SμV) [2]. SμV shields a software-based Trusted Computing Module (TCM) from untrusted application software using selective software virtualization and assembly-level code verification. However, due to the runtime safety checks, this approach introduces increased execution time. Moreover, SμV is considered memory-safe and crash-free, but it has not been fully verified yet. HAtt [1] is a hybrid remote attestation, which ensures the high availability of IoT devices during the software attestation process. The proposed attestation technique uses physical unclonable functions (PUFs) to protect the secrets of an IoT device from physical attacks. The security analysis shows that the proposed attestation technique can effectively detect roving malware. RATA [8] is a provably secure approach to detect transient malware that infects a device (by modifying its binary), performs its nefarious tasks, and erases itself before the next attestation. SARA [9] is an attestation protocol that aims to attest a large number of devices. SARA exploits asynchronous communication capabilities among IoT devices in order to attest to a distributed IoT service they execute. LIRA-V [19] is a lightweight system for performing remote attestation between constrained devices using the RISC-V architecture. It uses read-only memory and the RISC-V Physical Memory Protection (PMP) primitive to build a trust anchor for remote attestation and secure channel creation. SCRAPS [15] is a collective RA scheme that achieves scalability by outsourcing verifier duties to a smart contract and mitigates DoS attacks against both provers and verifiers.

Nevertheless, most of the works focus on the integrity of the lightweight embedded devices, and the proposed attestation protocols need to check the whole memory of the target devices. However, with the development of computer hardware, even some IoT devices are equipped with a large amount of memory and installed operating system. The memory of these devices is also at risk of malicious tampering. DRSA focuses on challenging the remote attestation schemes on devices with more processing power and more memory. By utilizing debug registers, DRSA can relocate itself and escape detection from existing software attestation schemes.

8 Conclusion and Future Work

In this paper, we presented DRSA, a new specific attack against existing software attestation techniques on the x86 platform. DRSA circumvents attestation by relocating itself and restoring the contents before it is measured. It leverages debug registers provided by the debugging architecture to monitor the beginning and the end of an RA measurement, erase itself, and restore the original program memory contents before attestation. We designed and implemented the prototype of DRSA under the Windows and Linux operating systems, respectively. From our experience, we can conclude that DRSA incurs very little attack overhead, which is difficult to detect by the existing time-based attestation schemes on the x86 platform. We also stress that an attacker can hide malicious code using debug registers, which is not taken into account by existing attestation schemes. We argue that attestation schemes should pay more attention to high-performance devices with debug architectures and more memory. For future work, we will investigate software-based attestation protocols that can guarantee security for devices with operating systems and more complex mechanisms.

References

1. Aman, M.N., et al.: HAtt: hybrid remote attestation for the internet of things with high availability. IEEE Internet Things J. **7**(8), 7220–7233 (2020)
2. Ammar, M., Crispo, B., Jacobs, B., Hughes, D., Daniels, W.: Sμv-the security microvisor: a formally-verified software-based security architecture for the internet of things. IEEE Trans. Dependable Secure Comput. **16**(5), 885–901 (2019)
3. Ammar, M., Crispo, B., Tsudik, G.: Simple: a remote attestation approach for resource-constrained IoT devices. In: 2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS), pp. 247–258. IEEE (2020)
4. ARM, A.: Security technology building a secure system using trustzone technology (white paper). ARM Limited (2009)
5. Brasser, F., El Mahjoub, B., Sadeghi, A.R., Wachsmann, C., Koeberl, P.: TyTAN: tiny trust anchor for tiny devices. In: Proceedings of the 52nd Annual Design Automation Conference, pp. 1–6 (2015)
6. Carpent, X., Rattanavipanon, N., Tsudik, G.: Remote attestation of IoT devices via smarm: shuffled measurements against roving malware. In: 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 9–16. IEEE (2018)
7. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 400–409 (2009)
8. De Oliveira Nunes, I., Jakkamsetti, S., Rattanavipanon, N., Tsudik, G.: On the TOCTOU problem in remote attestation. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 2921–2936 (2021)
9. Dushku, E., Rabbani, M.M., Conti, M., Mancini, L.V., Ranise, S.: SARA: secure asynchronous remote attestation for IoT systems. IEEE Trans. Inf. Forensics Secur. **15**, 3123–3136 (2020)

10. Eldefrawy, K., Rattanavipanon, N., Tsudik, G.: Hydra: hybrid design for remote attestation (using a formally verified microkernel). In: Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks, pp. 99–110 (2017)
11. Group, T.C.: Trusted platform module (TPM) (2017). http://www.trustedcomputinggroup.org
12. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 3B: System programming Guide, Part **2**(11), 0–40 (2011)
13. Hao, S., et al.: Deep reinforce learning for joint optimization of condition-based maintenance and spare ordering. Inf. Sci. **634**, 85–100 (2023)
14. Noorman, J., et al.: Sancus: low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 479–498 (2013)
15. Petzi, L., Yahya, A.E.B., Dmitrienko, A., Tsudik, G., Prantl, T., Kounev, S.: {SCRAPS}: scalable collective remote attestation for {Pub-Sub}{IoT} networks with untrusted proxy verifier. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 3485–3501 (2022)
16. Seshadri, A., Luk, M., Shi, E., Perrig, A., Van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, pp. 1–16 (2005)
17. Seshadri, A., Perrig, A., Van Doorn, L., Khosla, P.: SWATT: software-based attestation for embedded devices. In: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004, pp. 272–282. IEEE (2004)
18. Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.: Remote software-based attestation for wireless sensors. In: Molva, R., Tsudik, G., Westhoff, D. (eds.) ESAS 2005. LNCS, vol. 3813, pp. 27–41. Springer, Heidelberg (2005). https://doi.org/10.1007/11601494_3
19. Shepherd, C., Markantonakis, K., Jaloyan, G.A.: LIRA-V: lightweight remote attestation for constrained RISC-V devices. In: 2021 IEEE Security and Privacy Workshops (SPW), pp. 221–227. IEEE (2021)
20. Yang, X., et al.: Towards a low-cost remote memory attestation for the smart grid. Sensors **15**(8), 20799–20824 (2015)
21. Yang, Y., Wang, X., Zhu, S., Cao, G.: Distributed software-based attestation for node compromise detection in sensor networks. In: 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), pp. 219–230. IEEE (2007)
22. Zhang, N., Tan, Y.A., Yang, C., Li, Y.: Deep learning feature exploration for android malware detection. Appl. Soft Comput. **102**, 107069 (2021)
23. Zhang, Q., et al.: A hierarchical group key agreement protocol using orientable attributes for cloud computing. Inf. Sci. **480**, 55–69 (2019)
24. Zhu, H., Tan, Y.A., Zhu, L., Zhang, Q., Li, Y.: An efficient identity-based proxy blind signature for semioffline services. Wireless Commun. Mobile Comput. **2018**(2), 1–9 (2018)