Key-Hiding On The ARM Platform

By

Alexander Nilsson, Marcus Andersson and Stefan Axelsson

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2014 EU

Amsterdam, NL (May 7th - 9th)

Digital Investigation 11 (2014) S63–S67

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Key-hiding on the ARM platform

Alexander Nilsson, Marcus Andersson, Stefan Axelsson*

Blekinge Institute of Technology, Sweden

Keywords:
Cold-boot
Cryptography
Computer architecture
ARM

ABSTRACT

To combat the problem of encryption key recovery from main memory using *cold boot*-attacks, various solutions has been suggested, but most of these have been implemented on the x86 architecture, which is not prevalent in the smartphone market, where instead ARM dominates.

One existing solution does exist for the ARM architecture but it is limited to key sizes of 128 bits due to not being able to utilise the full width of the CPU registers used for key storage. We developed a test-implementation of CPU-bound key storage with 256-bit capacity, without using more hardware resources than the previous solution. We also show that access to the key can be restricted for programs executing outside the kernel space.

Introduction

Today, Full Disk Encryption (FDE) is available on modern mobile devices running OS:es such as iOS, Android and Windows Phone. Android is the most commonly used Operating system (Adamo, 2013), and as of version 4.0, FDE can be enabled provided the user uses a PIN or alphanumeric password.

When physical access to a device is available attacks such as *cold boot*-attacks can be performed (Halderman et al., 2009; Müller et al.) on the device in order to acquire the disk encryption key from main memory (RAM). Several ways to mitigate such attacks has been suggested (Müller et al., 2010; Müller et al., 2011; Simmons, 2011) however many of these are specific to the x86 architecture. Only one approach, called *ARMORED*, has focused on the ARM architecture (Götzfried and Müller).

Our work investigates the feasibility of extending solutions, such as *ARMORED*, to use key-sizes of up to 256-bits. The operating system that is being studied is Android 4.0.3 but primarily it is its Linux kernel (version 2.6.39.4 on our device) that is of importance.

In Section 2 we explore the related work done in this area (as stated, mainly on the x86 architecture), followed by our contributions. The technical details of our implementation is described in Section 3. In Section 4 we present our testing methods and the results of these tests. Sections 5 and 6 contain the conclusions and suggested future work respectively.

Background

Related work

Cold Boot-attacks (Halderman et al., 2009) are attacks against FDE solutions that aim to recover encryption keys from memory. They are possible due to the fact that memory content fades away slower the colder the RAM chips are. This makes it possible to reboot a computer without losing too much information and boot into an operating system which can read out the memory content.

FROST (Müller et al.) is a forensic tool that can recover encryption keys from memory on smartphones running the Android operating system with FDE enabled. Encryption keys are read from memory by performing a *cold boot*-attack on the smartphone and then flashing the FROST image onto the device which can then recover the encryption keys and decrypt the encrypted data partitions.

* Corresponding author.
E-mail addresses: alex.caelus@gmail.com (A. Nilsson), mban009@gmail.com (M. Andersson), stefan.axelsson@bth.se (S. Axelsson).

http://dx.doi.org/10.1016/j.diin.2014.03.008

AESSE (Müller et al., 2010) is a ”cold boot”-resistant implementation of AES for the Linux operating system on the x86 architecture. To prevent encryption keys from being acquired during memory attacks such as the *cold boot*-attack *AESSE* never stores keys in memory, but instead stores them directly in the SSE registers of the processor. *AESSE* also makes sure that no intermediate encryption state ever leaves the processor which would compromise the security of the implementation. However the security provided by *AESSE* comes at a high performance cost which makes it less attractive in practice.

TRESOR (Müller et al., 2011) (German for *safe* or *vault*) is the successor to *AESSE* and employs the same general principles as *AESSE* with substantially improved performance. Instead of using SSE registers for key storage, which caused a lot of compatibility problems, *TRESOR* uses the debug registers present in the x86 architecture. To increase the performance of the solution the *Intel AES-NI* instruction set for hardware accelerated encryption is used, which makes the implementation perform on par, or better than, the standard AES implementation, despite the fact that using CPU-registers precludes the common performance enhancing technique of performing (round) key expansion/key scheduling in main memory and reuse the expanded keys in subsequent encryption/decryption of blocks using the same key.

LoopAmnesia (Simmons, 2011) is a solution very similar to *TRESOR*, but instead of using debug registers LoopAmnesia uses performance counter registers. Another difference is that LoopAmnesia does not use the *Intel AES-NI* instruction set which makes the implementation slower than *TRESOR* and the standard AES implementation.

ARMORED (Götzfried and Müller) is an adaptation of TRESOR to the *ARM architecture* and the Android operating system. Due to technical limitations in the debug registers, only 128-bit encryption keys are supported by *ARMORED*, but since Android is currently limited to 128-bit keys for its disk encryption this was considered sufficient by the authors, and they did not further investigate whether these technical limitations could be addressed.[1]

Our contribution

Our contribution is an improvement of the storage solution employed in *ARMORED*, allowing us to store keys that are 256 bits in length. This was motivated by the following passage by Götzfried and Müller, where they discuss the drawback of not being able to use the full register width for key storage, limiting their maximum key size to 128 bits instead of 256 bits:

> “[Regarding ARM breakpoint registers] But unlike the debug registers in 64-bit x86 CPUs, they are too small to hold AES-256 keys. (Note that debug registers can be written into RAM due to context switching as well, but we specifically prohibit that by patching respective kernel routines.)
>
> On ARM, the least significant two bits of each 32-bit break- and watchpoint register are necessarily zero due to the memory alignment in ARM. Since instructions are consistently 32-bit wide, they are always located at 4-byte aligned addresses. Hence, the least significant two bits are omitted for setting break- and watchpoints because they must be zero anyway. *As a consequence, these bits are not available as key storage* [Our emphasis]. For the sake of convenience, we divided the key-sequence into 16-bit chunks; specifically, we use four breakpoint and four watchpoint registers, giving us a total of 8.16 = 128 bits as key storage. This is enough to accommodate AES-128, but not enough to accommodate AES-256. However, since Android's encryption feature is based on AES-128, this does not pose a problem.
>
> In future releases we could store more than 16 bits per register, and if we find additional break- and watchpoint registers, we could accommodate AES-256.”

This passage piqued our interest as to whether it would be possible to solve the problem of storing 256-bit AES-keys without having to use any additional break- and watchpoint registers.

The technical details of this part of the ARM architecture are as follows: the debug registers on the ARM architecture consist of four different kinds of registers; *breakpoint value* registers, *breakpoint control* registers, *watchpoint value* registers and *watchpoint control* registers. For each breakpoint or watchpoint value register there is a corresponding control register that controls the use of each value register.

The problem that prevented *ARMORED* from storing larger keys than 128 bits is the fact that the last two bits in each value register on the *ARM architecture* must be zero as ARM requires all executable code in memory to be 32-bit aligned, hence all instructions start on an address evenly divisible by four.

To overcome this problem we make use of every value register's corresponding control register and store the remaining two bits there (Fig. 1). This can of course only work if there are option bits available which can be used to store arbitrary data, without inadvertently negatively affecting the flow of execution when written. When investigating the actual semantics of the option bits in the control registers, we found that the first and second bit in both the watchpoint and breakpoint control registers are arbitrarily read- and writable (unlike most of the other 30 bits). These two bits are originally used for access control, e.g. controlling which privilege levels that trigger a breakpoint or watchpoint, but since bit zero has the value zero the breakpoint or watchpoint is disabled and therefore these bit values should not affect execution. Fortunately, our investigation shows that they can indeed be used freely in practice when the corresponding break-/watchpoint register is disabled.

To determine the validity of this approach, we created a system call which reads and writes 256 bits seamlessly to these registers. In total, four pairs of breakpoint value/control registers and four pairs of watchpoint value/control registers were used in the implementation.

[1] We would like to stress that we are only stating that this limitation *was not* addressed, not that it *could not* have been addressed.

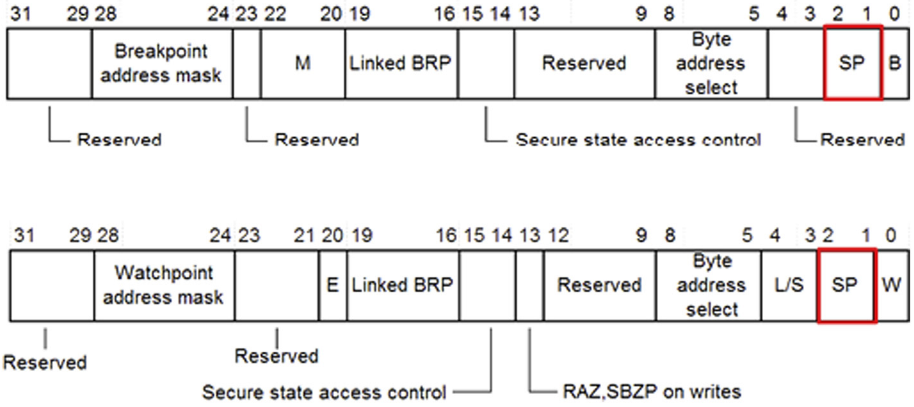

Fig. 1. Breakpoint and watchpoint control registers.

Implementation

To test our idea we implemented both a system call and a user space application. We will give a more detailed description on both of these in Sections 3.1 and 3.2 respectively.

Additionally we must also be able to keep the hardware debug registers from being read and tampered with by code running in user space. A more detailed description of preventing this kind of access is given in Section 3.3.

Code running in kernel space cannot be prevented from accessing these registers and thus we must keep any malicious user from ever gaining such privilege in the first place. How to actually keep such privilege escalation from happening is outside the scope of this paper, a good start however is to disable Loadable Kernel Modules and the KMEM device.

In order to test our implementation in as realistic an environment as possible, we also added a call to our implementation from the dm-crypt Fruhwitch, 2005 'crypto_cipher_setkey' function which is in turn called by the android boot process. Additional detail is provided in Section 3.4.

System call

This paper is not intended to be a tutorial on how to implement a system call in the ARM specific branch of the Linux kernel, we will however try to give as clear a picture as we can in order to increase the reproducibility of our experiments. Hence we will include a list of all files that we have touched in order to make the kernel compile the system call.

arch/arm/include/asm/unistd.h

This file contains a list of all system call numbers, in this instance the next number available to us was 374 and we defined it as:

```
"#define __NR_key_research (__NR_SYSCALL_BASE+374)".
```

`arch/arm/kernel/calls`

This file is where we actually declare our system call and provides the linker with the information required in order to link our system call that we have defined in the following next file.

We appended the following to the list of system calls: "`CALL(sys_key_research)`"

`arch/arm/kernel/key_research.c`

This is the file we created to house our implementation. Our system call uses the following prototype:

```
"int sys_key_research(int mode, char* data);"
```

where `mode` indicates read or write and `data` obviously is the key buffer from/in which to read/write the key.

Below we see a reduced example of the write function which only stores the first 32 bits of the key. The full implementation simply duplicates each step for the rest of the key.

```
//Add first 30 bits of data to temporary
    register
register long r0 asm("r0") = ((u32*)data)
    [0] & 0xFFFFFFFC;

//Write data to debug register
__asm__("MCR p14, #0, r0, c0, c0, #4" ::
    "r"(r0) :);

//Read current values of control
    registers
__asm__("MRC p14, #0, r0, c0, c0, #5" : "
    r"(r0) ::);

//Write last 2 bits to the temporary
    register
r0 = (r0 & (~0x6)) | ((((u32*)data)[0] &
    0x3) >> 1);

//Write back to control registers
__asm__("MCR p14, #0, r0, c0, c0, #5" ::
    "r"(r0) : );
```

The read function simply does the reverse of the write function and it is therefore not included in this paper.

include/linux/key_research.h

In this header file we declare our function and a simple c-wrapper so that we can call the system call in our user-space application without using the special swi instruction. This function will be compiled into whatever program includes this file.

```
void key_research(int mode, char* data)
{
        register int r0 asm("r0");
        register char* r2 asm("r2");
        register long r7 asm("r7");

        r0 = mode; //first argument
        r2 = data; //second argument
        r7 = 374; //syscall number

        asm volatile("swi 0x0"
                :"=r"(r0), "=r"(r2), "=r"
                   (r7)
                :"r"(r0)
                :"memory");
};
```

arch/arm/kernel/Makefile

Of course we also had to add the key_research.c file to the compile process, we did this by simply appending key_research.o to the object file list (obj-y).

User space application

As stated above we also implemented a very simple user space application that utilises the key_research system call. There is nothing special about this program; with no arguments it calls the system call in read mode and prints the resulting 256-bit value as a hexadecimal string.

If given an argument it writes the value of the argument (interpreted as a hex string and converted to bytes) to the debug registers with the system call in write mode.

Prevent access to HW-regs

In order to prevent other user-space programs from accessing the debug registers we must disable the ptrace system call's support of hardware registers (in arch/arm/kernel/ptrace.c and arch/arm/kernel/hw_breakpoint.c). This is done by compiling the kernel with the configuration variable CONFIG_HAVE_HW_BREAKPOINT undefined.

To undefine this configuration variable one must edit the arch/arm/Kconfig file and remove the logic that overwrites .config with CONFIG_HAVE_HW_BREAKPOINT=y on each compile.

As previously mentioned the above method only disables access to the debug registers from user space, any code running under kernel space has complete access to these registers.

To try and ascertain whether the rest of the kernel contains any instructions or code that could reference these registers we performed a *grep* of the kernel source code. We found no files other than the ptrace.c, hw_breakpoint.c and key_research.c that contain code that writes or reads from/to these registers.

Adding a key during android bootup

During our experiments we want to have an environment that emulates a real usage scenario as closely as possible. With this goal in mind we added a call to our system call (in write mode) from the include/linux/crypto.h file, specifically in the crypto_cipher_setkey function.

The above function is called by dm-crypt, which in turn is called by the android boot-up process if disk encryption is activated.

Testing & results

Verification of preventing access to HW-regs by code review

To ensure there was no other code in the Linux kernel that accesses the registers, other than what we have presented so far, we made a simple search of the entire Linux kernel's source code with the following commands:

```
grep -R -i 'MCR p14' .

grep -R -i 'MRC p14' .
```

In the full output we saw a few previously unknown files that used assembly to write to the same co-processor (p14) as the debug registers, but upon closer examination these were all false alarms. None of these hits actually write to our registers of interest.

According to our understanding of the ARM manual there is no way to access the values of the debug registers other than using the MCR and MRC instructions, so we feel relatively confident that there are no other places in the kernel that accesses these registers other than the two that were previously mentioned.

Key integrity

In order to test our solution and the integrity of the keys, we wrote easily distinguishable values into the registers using our system call and then subjected the device to usage which would normally write data to the debug registers. These usage tests included things such as explicitly setting hardware break and watchpoints with gdb and general adb usage. After these tests were conducted we read the debug register values again with our system call and compared them to the values we had originally written to them.

During our tests we never managed to overwrite any key values written into the debug registers. Of course this does not mean it is not possible to in other ways write data to the debug registers, but in our tests we feel that we have covered all normal use cases which concerns debug registers.

It should be noted that since debuggers in general cannot rely on an architecture having breakpoint registers, or that available breakpoint registers may already be occupied, they will in general work as advertised, typically using software breakpoints instead of hardware breakpoints when they are unavailable.

On-boot key insertion

To test this function we enabled FDE on the device, which would enable the `dm-crypt` module we had modified to run on boot. We then rebooted the device a number of times and each time read out the value located in the debug registers. After comparing these values to the one we specified to be written on boot we found that they were equal on every occasion and concluded that the on-boot key insertion function was working properly.

Conclusion

We have demonstrated that it is indeed possible to store a 256-bit key on the processor chip on the ARMv7 architecture by using only eight 32-bit break- and watchpoint registers. We have also determined that it is possible to (a certain extent) prevent access the key from userspace. There is however no way of preventing an attacker from changing or reading out the values if he/she has the means to inject code into kernel space and execute it.

Therefore we strongly recommend that any implementation that uses CPU-bound storage of encryption keys disable LKMs and the KMEM device at minimum. Other ways of preventing access to the kernel is outside the scope of this paper.

Future work

More extensive testing and using different kernel versions and other hardware may be required before any serious guarantees can be made regarding the security of CPU-bound key storage used outside the academic community.

We have not conducted any performance testing of our solution due to the fact that there is no encryption scheme that is currently compatible with our prototype implementation. *ARMORED* comes the closest but some work is required to modify our solution to work with theirs. One way of minimising the work required to conduct performance testing would be to only utilise the first 128 bits of our key storage solution and thus we would not require to implement the full 256-bit AES algorithm. The performance could then be directly compared between the two key storage solutions. However, as our approach only adds a small handful of extra instructions to the key setup, and since the execution time is dominated by the actual encryption (especially for AES as the number of crypto rounds increase with increased key lengths), we conjecture that the impact should be slight.

Since our solution only focuses on the storage of encryption keys, and not any actual encryption, the next natural step would be to extend to an already existing solution like *ARMORED* to support the full 256-bit AES algorithm.

To make this kind of system applicable in practice a patch which allows Android to safely derive the key from the PIN-code/password and write the it to the processor without leaving any traces in memory is required and we consider this the next step after applying *ARMORED* in either the 128-bit vanilla mode or with 198/256-bit modification.

Further research is required to make our solution support the ARMv8 (64-bit) architecture, we feel that this should not pose a problem however since that architecture requires the registers to be twice the size of the 32-bit ARMv7 architecture.

References

Adamo S. Comscore reports july 2013 U.S. smartphone subscriber market share. Online at, http://www.comscore.com; Sep. 2013. Accessed at time of publication.

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold-boot attacks on encryption keys. Communications of the ACM 2009;52(5):91–8.

T. Müller T, M. Spreitzenbarth M, F. C. Freiling FC, . Forensic recovery of scrambled telephones.

Müller T, Dewald A, Freiling FC. Aesse: a cold-boot resistant implementation of aes. ACM. In: Proceedings of the Third European Workshop on System Security; 2010. pp. 42–7.

Müller T, Freiling FC, Dewald A. Tresor runs encryption securely outside ram. In: USENIX Security Symposium; 2011.

Simmons P. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. ACM. In: Proceedings of the 27th Annual Computer Security Applications Conference; 2011. pp. 73–82.

J. Götzfried J, T. Müller T, . Cpu-bound encryption for android-driven arm devices.

Fruhwirth C. New methods in hard disk encryption. Master's thesis. Vienna, Austria: Institute for Computer Languages Theory and Logic Group Vienna University of Technology; July 2005.