UEFI BIOS and Intel Management Engine Attack Vectors and Vulnerabilities

Alexander Ogolyuk, Andrey Sheglov, Konstantin Sheglov
Saint Petersburg National Research University of Information Technologies, Mechanics and Optics
St. Petersburg, Russia
xms2007, npp-itb@yandex.ru

***Abstract*—we describe principles and implementation details of UEFI BIOS attacks and vulnerabilities, suggesting the possible security enhancement approaches. We describe the hidden Intel Management Engine implementation details and possible consequences of its security possible discredit. Described breaches in UEFI and Intel Management Engine could possibly lead to the invention of "invulnerable" malicious applications. We highlight the base principles and actual state of Management Engine (which is a part of UEFI BIOS firmware) and its attack vectors using reverse engineering techniques.**

I. INTRODUCTION

The Intel Management Engine is the key part of x86 platform architecture and the big part of the modern computers UEFI BIOS subsystem. This system is mostly hidden from user or administrator access. It includes secured and privileged executable code which can be accessed or controlled from normal operating system environment. Even many security experts don't know (or don't know much) of its existence. In the year 2006 Intel introduced the basic AMT (Active Management Technology) subsystem which was the remote management solution for Intel based computers (and servers). It included:

- inventory services,
- update service,
- management,
- diagnostics
- remote access services.

This subsystem was implemented not only in Intel based server firmware like in previous generations of remote access technologies (including IPMI – Intel platform management Interface) but in all desktop computers. To implement this subsystem all AMT compatible computers have additional microcontroller integrated into Intel chipsets. AMT subsystem introduced many new features (which are "outstanding" from security point of view). They were:

- embedded HTTP(S) server,
- out of band access to integrated network adapter including the control of all network incoming and outgoing packets,
- access to all input and output devices,
- access to NVRAM and many more.

This microcontroller (and the whole AMT subsystem) starts to work even without user pressing the computer power button, just the power adaptor needs to be on, i.e. it works even in computer switched off mode. Later AMT became a part of Intel Management Engine subsystem. In 2007 Intel introduced newer subsystem features:

- full RAM DMA access
- direct access to the integrated video adaptor memory (which makes full screen grabbing in real time possible)
- KVM standard remote access, etc.

During next years many standard BIOS features like:

- Integrated Clock Control,
- ACPI (power interface),
- TPM (trusted platform module)

migrated from main BIOS firmware into Management Engine subsystem (which is also placed inside firmware SPI cheap). Originally this subsystem was supported only on high-end Intel motherboards which were not too popular because of high prices. But after years 2010-2011 this subsystem was introduced in all Intel chipsets both in server, desktop and mobile segments (notebooks, tablets and smart phones). Originally Intel used SPARC and ARC32 based microcontrollers. In modern systems Intel replaced this hardware with x86 based controllers. This makes reverse engineering of Management Engine subsystem code easier for average attacker. Current implementation of Intel Management Engine includes the following components:

- Host microcontroller (includes integrated ROM)
- Firmware SPI cheap, which is partially co-used by main UEFI BIOS firmware
- Dedicated RAM (about 32 Mb)
- UEFI BIOS DXE/SME modules

ISSN 2305-7254

- Management Engine Interface (main CPU hosted code interface to controller)
- Dedicated network controller for direct Ethernet adaptor access

Resuming all above we can say that Intel Management Engine introduces the new level of x86 code execution (additionally to well known ring 0-3 of basic i386 architecture and SMM or Hypervisor levels). Code executed on this level (ME) is fully hidden and can't be controlled from all other execution levels (including OS code or BIOS code). Management Engine controller architecture is very complex and goes beyond our scope.

We will describe mostly software implementation. As we said above Management Engine subsystem resides in SPI firmware cheap. Standard Intel Firmware includes following regions:

- Descriptor, which describes all SPI memory regions and their access attributes
- UEFI BIOS firmware
- Management Engine firmware
- GbE (Ethernet network adaptor firmware)
- PDR (Vendor specific extra modules)

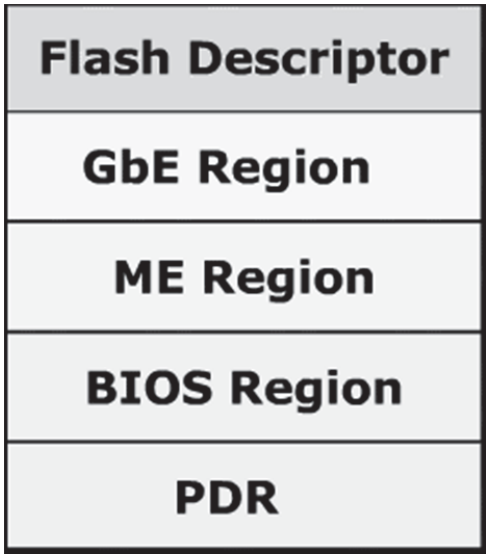

Fig. 1. Firmware regions

The code segment of Management Engine subsystem is protected with digital signing. It includes Intel RSA public key. Intel protects the code with own private key and anyone can verify the signature with included public key. The attacker can't replace the included public key with he's own because the signature and integrity are verified by boot code (situated in microcontroller ROM). The Management Engine code is additionally protected by read only memory region attribute which prevents its overwriting. Management Engine subsystem works in two modes: privileged and user mode. In privileged mode the subsystem code can access all hardware devices and memory (feature described above). All subsystem code is divided into modules. The main modules are controller Operating System kernel and drivers. Also there are services modules (like AMT and others) and non privileged modules.

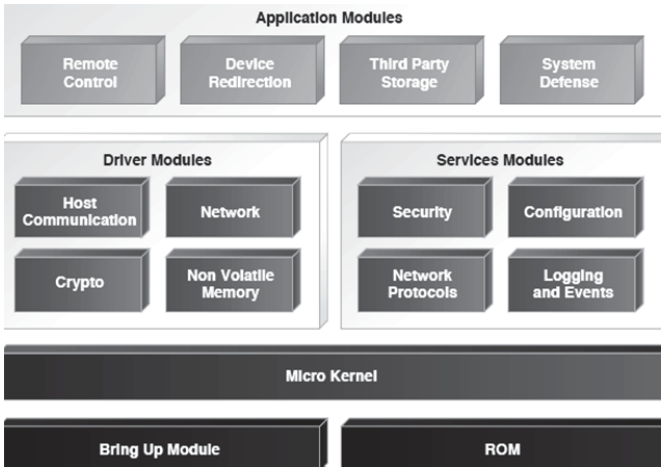

Fig. 2. Intel Management Engine subsystem modules architecture

Looking on this Intel Management Engine subsystem architecture we can definitely say that any compromise of its security could lead to huge consequences. If attacker could rewrite or insert own code into Management Engine subsystem code segment he can get many benefits like:

- Fully invisible to Operating system and BIOS malicious application
- Permanent malicious code residence
- No detection feature of the base malicious application
- Full restore of end dropper code (executed inside Operating System) by the base malicious application resided in firmware
- Full access to all RAM, devices, video adaptor and Ethernet adaptor (to log all computer activity)
- Privileged real time execution
- Software SPI rewrite protection

There is no known attacks implementation on Intel Management Engine subsystem so far, but it doesn't mean it is fully safe. Also there is a basic vulnerability which is Intel private key leakage. If such key will be available to the attacker there will be no protection against overwriting or inserting malicious code into Management Engine subsystem code segment. The basic way of the write protection of Management Engine memory region bypassing could be hardware SPI programming (with chip hardware flasher like CH341A) or even using software methods described below.

The other problem and attack vector which will be discussed more detail below is Management Engine subsystem integration with the main UEFI BIOS and related vulnerabilities which are already more actual and practically approved.

II. INTEL MANAGEMENT ENGINE SUBSYSTEM ATTACK VECTORS

As we said there are no known successful attack implementations for Management engine subsystem, but we can list the following theoretical approaches:

- ME dedicated memory dump by disabling memory lock bits and memory region attributes (needs initialization routines reverse engineering)
- "Cold boot" attack – RAM modules swap to read or replace the contents (need slow speed memory)
- Change ME dedicated memory size while initialization phase enlarging it which will make ME use the top part and remove enlarging on the following boot to allow access to the previously written top region.
- Use self written Java applets asking ME subsystem to execute them (one of the AMT features)
- Reverse engineering Intel Windows applications (C#, C++, Java) which can communicate to ME subsystem

The read RAM/ROM like attacks (Cold boot, etc.) are needed to read the dedicated memory (chip based) which is not accessible by other means and contains portions of ME code needed to fully reverse engineering of ME subsystem algorithms)

These approaches are not approved yet. But we plan to try and research them, which can bring potential vulnerabilities discovering in the Intel Management Engine subsystem.

The other reason to worry about Intel Management Engine subsystem is its wide usage in modern computers. Intel declares all remote access (AMT) features are switched off on most of computers (except high-end and server platforms). But the reversing of firmware (all Intel based platforms use Intel code) shows that the code itself is present on all computers. It is just not activated.

But there is no any guarantee that some "magic" password or network packet can't make this code active (even if this request is authorized by some US federal service officer) making your system to start spy for you or to work under remote management. And the most dangerous thing is that it is near impossible to change this situation. Intel code is fully closed and never will be available for review (to prove security mechanisms and to prove unauthorized "bookmark" absence). The only alternate x86 compatible platform is AMD, but it is much less widespread and has own similar subsystems. Incompatible platforms (like ARM, SPARC, etc) do exist but they don't have any visible percent of usage in desktop and laptop markets (only tablet and smart phone markets widely use ARM architecture). The end firmware modification (by the owner) to remove unnecessary AMT (remote access and hardware control) code blocks is also not possible because of integrity checks and RSA signing described above.

III. UEFI BIOS ATTACK VECTORS

More known part of the Intel based computers is UEFI BIOS firmware. EFI (*Extensible Firmware Interface*) standard came to replace old BIOS (*Basic Input Output System*) in 2004 – 2006.

First EFI was introduced for IA64 platform then became the standard for all x86 based platforms and many others too (ARM based). EFI is widely used on x86 computers since 2008-2009, on Apple computers since 2007. One of the benefits of EFI code is that is mostly written on C (unlike assembler for BIOS) and has many features and uses known formats. All its code is stored in non-volatile memory (SPI flash chip) and it is the first code which runs after system turns on.

Unlike BIOS code, EFI (UEFI) code runs in 32bit protected mode.

UEFI BIOS boot phases are:

- SEC (security)
- PEI (Pre EFI)
- DXE (Driver Execution Environment)
- BDS (Boot Device Select)
- TSL (Transient System Load)
- RT (Run Time, i.e. OS code execution)
- AL (After Life, i.e. shutdown)

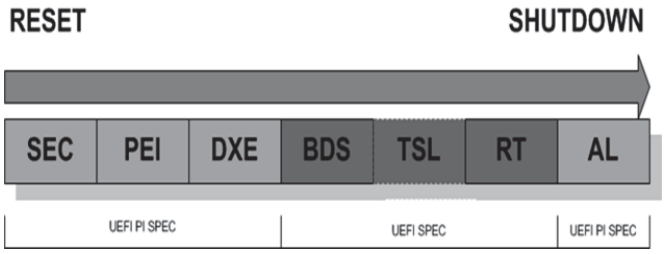

Fig. 3. UEFI boot phases [7]

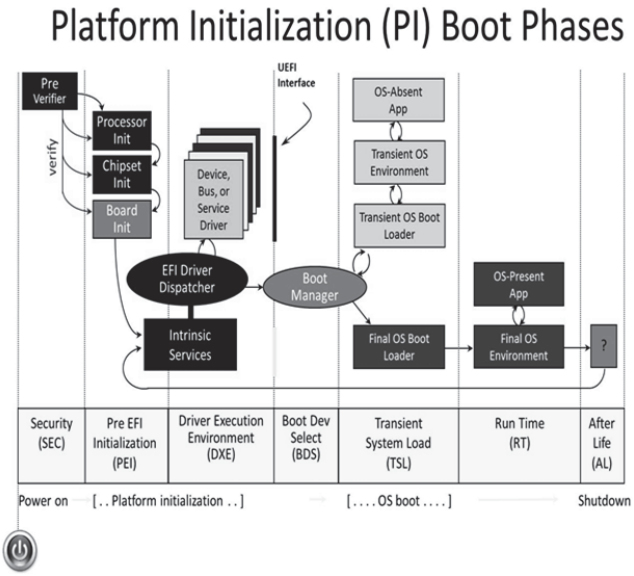

Fig. 4. UEFI Boot Phases interaction scheme [7]

The system boot runs through *SEC* (security phase). During this phase temporary memory initialization is performed and firmware integrity can be checked, plus PEI phase initialization is prepared.

Then PEI (*Pre EFI Initialization*) phase starts which serves similar purposes as old BIOS initialization phase (RAM init, ROM to RAM copy, PEI modules execution, interfaces initialization, preparing to go into DXE phase if system is not in Sleep Resume state, else executing S3 boot script, etc.).

After this phase UEFI builds the structured DXE space *(Driver Execution Environment*) to run UEFI drivers and services. Both DXE drivers and services can have dependencies (DXE executables) which also need to be loaded. DXE phase also performs hardware initialization and building hardware access abstract interface (for services). Unlike MBR used in BIOS times it uses *EFI System Partition* (FAT32). UEFI tries to find boot code on EFI partitions and give execution to this boot code. If nothing found the error message is displayed els the Operating system is loaded. System bootloader, operating system drivers and services can contact the firmware via specially designed protocols. After full loading operating system can access some of this interfaces via the *EFI runtime services*. DXE phase includes SMM sub phase (*System Management Mode)* which is described in more detail in next chapter and [5].

As we can see UEFI BIOS firmware code is also a base key of computer security system (including Secure Boot mechanism which prevents unauthorized bootloaders to be executed). The attackers which will insert own code into UEFI BIOS code segments will receive such benefits:

- Persistent malicious code living
- Surviving OS uninstall, reinstall, disk formatting
- Getting access to hardware, RAM, CPU, Ethernet and video adaptors
- Hard software (and even hardware) detection by antivirus tools

The main attack vector is firmware SPI flash chip. If there is no integrity check for UEFI BIOS executable modules then inserting own code is easy with hardware chip programming (which is not very useful for attackers but still useful for government services). If such integrity checks are present attacker can disable them (by patching integrity check code block). In the following sections we will describe all modern available attack vectors including fully software implementations (which need no any hardware SPI programming but just a small piece of code executing).

The secondary attack vectors could be BIOS Setup which controls:

- BIOS region attributes (SPI Lock Bit)
- Secure Boot mechanism
- NVRAM settings
- BIOS password protection

The practically implemented approaches to attack BIOS Setup is changing NVRAM “SETUP” variable. This can be done through UEFI shell (build from Tiano-Core open sources or obtained from platform vendor or available in recovery boot images for target platform). After booting into UEFI shell application attacker can change SETUP variable (with special self written EFI application or shell commands like “set var”). The SETUP variable contains many critical system variables inside (it uses large memory region) including SPI lock bit (which denies firmware software overwriting). Lock bit variable address (which is needed to change it) is specific for end platform (and computer model) but can be reverse engineered from the BIOS image (available at the vendor web site).

Inside the main attack vector (firmware overwrites or code insert) there are several cases:

- No UEFI BIOS region protection at all - many system vendors don’t use read only attribute of BIOS region allowing writing into it (which makes BIOS code modification easy just with ordinary software flasher application). This case is especially probable on older systems (before 2013-2014)
- Protected UEFI BIOS region. In this case some specific vendor utilities (applications) can be used to disable protection or other methods used (described below).

BIOS region protection depends from BIOS vendor (manufacturer). All UEFI BIOS vendors use Intel code examples but other implementation details can differ. The most known UEFI BIOS manufacturers are:

- Intel
- AMI (desktop and laptop)
- Phoenix (desktop)
- Insyde hydrogen (laptop)

Most of computer vendors like HP, DELL, Lenovo, ASUS, GIGABYTE, ASROCK, MSI, Samsung, Sony, ACER, etc. do use their code (with small injections of vendor specific code).

The easiest way to rewrite UEFI BIOS region is the use of vendor flash utilities or rescue disks. Such software is officially “unavailable” but can be downloaded from unofficial BIOS web forums like:

- www.bios-mods.com
- www.insanelymac.com
- www.win-raid.com
- forums.mydigitallife.info

Also UEFI BIOS software is available even on official vendor websites (Lenovo, Hewlett Packard, ACER, Samsung, etc) inside so called rescue disks which are designed to repair computers after BIOS failure (during firmware upgrade or other reason).

The Intel Management Environment System Tools and Intel Flasher utility are the most known tools because they

mostly bypass firmware image integrity checks and just write the image into firmware region (ME or UEFI BIOS region). The disadvantage of this tool is the platform specific versioning. Each Intel chipset needs corresponding flasher utility executable. Modern existing versions (all available for download) are:

- *Intel ME System Tools v11.6 r5* - (Updated: 08/02/2017)
  For 100/200-series systems which come with ME firmware v11.0-11.6
- *Intel ME System Tools v11.0 r2* - (Updated: 29/01/2017)
  For 100-series systems which come with ME firmware v11.0
- *Intel ME System Tools v10 r1* - (Updated: 13/10/2016)
  For Broadwell mobile systems which come with ME firmware v10.0
- *Intel ME System Tools v9.5 r1* - (Updated: 13/10/2016)
  For 8-series systems which come with ME firmware v9.5
- *Intel ME System Tools v9.1 r1* - (Updated: 13/10/2016)
  For 8/9-series systems which come with ME firmware v9.1
- *Intel ME System Tools v9.0 r1* - (Updated: 13/10/2016)
  For 8-series systems which come with ME firmware v9.0
- *Intel ME System Tools v8 r1* - (Updated: 13/10/2016)
  For 7-Series systems which come with ME firmware v8
- *Intel ME System Tools v7 r1* - (Updated: 13/10/2016)
  For 6-series systems which come with ME firmware v7 *Intel ME System Tools v6 1.5MB/5MB r1* - (Updated: 13/10/2016)
  For 5-series (Ibex Peak) systems which come with ME
- *Intel ME System Tools v6 Ignition r1* - (Updated: 13/10/2016)
  For 5-series (Ibex Peak) or 89xx-series (Cave/Coleto Creek) systems which come with ME Ignition firmware v6
- *Intel ME System Tools v5 r1* - (Updated: 13/10/2016)
  For ICH10 systems which come with ME firmware v5
- *Intel ME System Tools v4 r1* - (Updated: 13/10/2016)
  For ICH9M systems which come with ME firmware v4
- *Intel ME System Tools v3 r1* - (Updated: 13/10/2016)
  For ICH9 systems which come with ME firmware v3
- *Intel ME System Tools v2 r1* - (Updated: 13/10/2016)
  For ICH8 & ICH8M systems which come with ME firmware v2

To recognize the needed Tools version the open source "ME Analyzer" application could be used. So using Intel ME tools there are less problems to write modified firmware to the SPI flash (most other vendors like *Insyde Hydrogen* do verify image CRC checksum and signatures inside image regions). And if no UEFI write protection lock is enabled (such case exists in many configurations as we described above) the modified firmware (with malicious code injected) can be flashed to the SPI chip by software method (using Itntel ME Tools). This image "upgrade" could be automated. More to say this "upgrade" can be performed not just from DOS or UEFI Shell, but from Windows itself (Intel has executable for all operating systems environments, including Linux).

If the UEFI BIOS region is write protected (BIOS Lock Enable Bit set) "set var" NVRAM variable and BIOS STUP attack vectors can be used (described above).

If there is no ME Tools available for Windows Operating system then UEFI Shell (from vendor rescue disk) or DOS bootable media can be used to run Intel ME Tools EFI or MS-DOS executables. In this case the automation of software attack is more complicated (needs multiply reboots) but is still possible.

IV. OTHER FIRMWARE ATTACK VECTORS

There are several other firmware attack vectors (additionally to Intel Management Engine and UEFI BIOS targeted attacks). They could be:

- *SMM (*System Management Mode) *code injection*
- *PCI device firmware (Ethernet, Video, Thunderbolt, etc.) code injection*
- *Secure Boot mechanism disabling*
- *Vendor specific*

SMM is another one privileged mode of code execution on x86 platform (starts from i486). SMM mode is activated by triggering SMI interrupt (can be hardware or software activated). Software SMI is generated by:

- *USB controller (in USN legacy mode)*
- *Intel Management Engine subsystem*
- *GPIO registers*
- *SMI Timer*
- *Chipset SMI (on IO port access)*
- *ACPI SMI (Sleep modes, etc.)*

After going into SMM mode CPU saves all context (including registers). By default SMM code can access all RAM (read and write access) and access to all connected devices. In the same moment SMM code is not accessible from OS (OS can only see SMM mode was called and nothing more).

So SMM mode (code execution) is a good target for attacker. The main attack vector for SMI is to try to generate software SMI interrupt from user code (this needs privileges to execute IO port communication commands like in, out) and try to find vulnerabilities in executed SMM code exploiting it to get some benefits. One of first known SMM attacks was SMM poisoning which consists in writing own malware code into

cache and triggering its execution by SMI interrupt (now this vulnerability is closed). Modern SMM attacks are based on changing SMM support code situated in RAM (i.e. not in secured SMM dedicated SRAM). Inserting code there and triggering SMO interrupt theoretically allows to execute own malicious code in SMM mode (with all SMM mode benefits). Most of systems designed before 2015 (when Intel introduced recommendations for attack prevention) are theoretically vulnerable to such approach.

Other SMM mode based technique can be DMA copy to SMM region. This attack vector more detailed description is very complex and goes beyond this article and will be described in other works together with PCI firmware attacks (basically consisted in firmware code injections) and ACPI attacks.

Secure Boot and Vendor specific are the last attack modes to describe in this article. UEFI BIOS Secure Boot mode prevents unauthorized boot code execution (OS bootloader). The boot code integrity and signing is checked using public keys stored in NVRAM (by default only Microsoft keys are present , allowing to load only Microsoft Windows bootloader code).

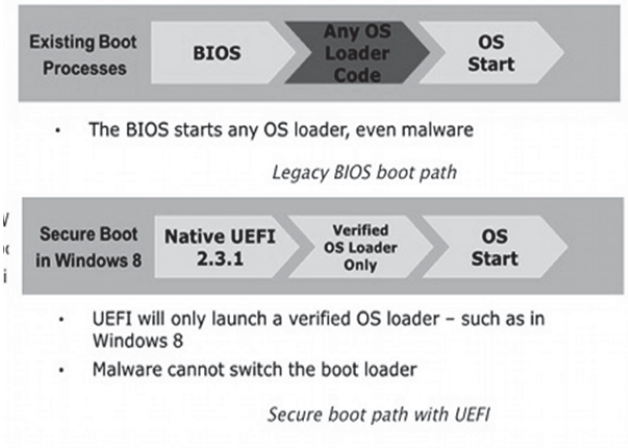

Fig. 5. Secure Boot Mode comparing to ordinary OS loading scheme

Theoretically the malicious boot code can be signed by Microsoft private key (Official certificate with generated keys can be purchased from Microsoft partners like Digicert, verisign, etc.) to bypass the secure Boot verification. The known non Microsoft boot loader which works with Secure Boot enabled on all computers is Canonical Ubuntu Linux boot loader.

The other option to disable Secure Boot mode is erasing NVRAM variables (from UEFI shell or even from Microsoft Windows) which store this keys. If UEFI BIOS code found no public keys it disables the Secure Boot Mode. This erasing operation also can be automated (i.e. performed by malicious software).

Vendor specific theoretical attack vectors are:

- SMM embedded flasher support code
- Hardcoded factory passwords (BIOS)
- Hidden write enable bits or variables
- Vendor specific code vulnerabilities

This last attack vectors are hard to research because of major difference in vendor codes and code versions inside one vendor platform. Also it needs deep reverse engineering to find them. But anyway this approach still can be used and researched in specific cases (targeting specific platform) and practically implemented.

V. CONCLUSION: PROTECTION IDEAS AND ROADMAP FOR FUTURE RESEARCHES

We must to admit that there is no easy and practically implemented universal solution for vulnerabilities and attack vectors described above. This is due to that Intel Platform and all firmware codes are not open sourced so can't be fully researched even using reverse engineering techniques. This can't be changed in near future (there are no alternatives to Intel and AMD x86 platforms). But we can list the possible directions in which community could go to solve this situation (and even implement some protection solutions without Intel support which one is very low possible). This directions could be:

- Disable all SMM code (if possible by patching or other methods)
- Disable any external firmware components (PCI boot)
- Disable S3 Bootscript (after sleep mode)
- SMI transaction Monitor extensive usage (to find malicious SMI calls)
- Enable Secure Boot mode
- Enable BIOS password
- Extensive reverse engineering of vendor's firmware samples to find and report vulnerabilities
- Code reviews (of open sourced UEFI based systems like Tiano-Core)

We plan to research more on these approaches in future works.

REFERENCES

[1] A.U. Sheglov, K.A. Sheglov, Informational systems security analysis and design. St.Petersburg: Professionalnaya literatura, 2017.
[2] A. Kumar, Active Platform Management Demystified: Unleashing the Power of Intel VPro Technology, New York: Intel Press, 2009
[3] X. Ruan, Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine, New York: APress, 2014.
[4] M. Rothman, G. Xing, Y. Wang, J. Gong "Reducing platform boot Time", Intel white papers, 2011, pp 1-42,
[5] Intel official site, AMT-SDK, Web: https://software.intel.com/en-us/amt-sdk
[6] Winraid official site, Firmware Drivers, Web: http://www.win-raid.com/t596f39-Intel-Management-Engine-Drivers-Firmware-amp-System-Tools.html
[7] Phoenix official site, UEFI specification, Web: http://blogs.phoenix.com/phoenix_technologies_bios/uefi/