Fuzzing Low Level Design – BIOS Guard

Yevgeniy Kiveisha

*SW Engineer, Flex Services Application Software & Technology*

[Yevgeniy.kiveisha@intel.com](mailto:yevgeniy.kiveisha@intel.com)

David Kaplan

*SW Engineer, Flex Services Application Software & Technology*

[David.kaplan@intel.com](mailto:david.kaplan@intel.com)

**Keywords**: Fuzzing, UEFI, BIOS, Python, Penetration test

# ABSTRACT

Security in Intel is an important aspect in almost every part of the product development. Whether the product is corporation internal or global, the security is very important in order to maintain Intel's integrity. Fuzzing takes a significant part of security SDL as the final step of security validation (S4). Fuzzing can be used in high level applications as well as in low level applications, however the complexity of doing this differs from one level to another. Low level fuzzing requires building your own tools in order to perform proper fuzzing while in high level fuzzing there are existing tools that can be used so in this case the fuzzing is much more flexible. In low level fuzzing the access to the application is not as obvious as in high level applications, moreover the testing environment (not so colorful..) will make your life a bit harder.

In this abstract the focus will be on fuzzing low level applications, meaning the environment is UEFI (Unified Extensible Firmware Interface) and has NO Operating System - not much of a friendly environment.

The problem is that the access to this environment is very limited and the most difficult part is that you have no OS to execute your script, custom application or any other fuzzing tool. So the imagination should take control to help you overcome this obstacle.

BIOS Guard is the new security technology that was released with Haswell CPU and its main purpose is to protect the BIOS from malicious BIOS updates. This techonolgy has definately added a security value to Intel's products and their integrity.

BIOS Guard is a good example of the module that was fuzzed under all described unfriendly environment limitations and the fuzzing process will be explained in this abstract.

The paper will try to show in a nutshell the fuzzing in the embedded world and share the approach that was used in order to get the results as well as sharing experience that was gained during the planning of this task execution.

This abstract will not discuss any of the details on how the module works since this project is considered as Intel Top Secret.

# INTRODUCTION

BIOS Guard is a core module, if it’s enabled its integrity is crucial to correct execution of the BIOS upload after the BIOS is being updated. If there was even a small crack in the BIOS Guard module security that would’ve allowed malicious users to update the BIOS with malicious code, this could’ve harmed Intel's integrity and Intel products. That's why fuzzing takes a big part of the penetration tests.

The problem is that BIOS Guard is being executed without OS, which makes it very hard to fuzz the module with scripts and fuzzing tools using conventional methods as with OS. Before creating our work plan for fuzzing we had to deal with a couple of questions:

* How do we create the mutated data that will be passed to the test framework?
* How do we execute BIOS Guard when we need, there is no executable files?
* How do we get the results and save them somewhere?
* What tools we should use?
* How do we create a tool that will automatically iterate through each an every piece of our mutated data?

The answers to the above questions will be discussed in this paper.

This abstract will describe the tools that were used during the task and at what point each of the tools was used.

One important clarification should be made, the fuzzing for this module was done based on white box approach this made it a bit easer because prior to the fuzzing, the security researcher had to learn the architecture and have extensive knowledge on how the module behaves.

Below is the list of the tools that were used during the fuzzing process:

* Python – was used for parsing the responses that were received from the fuzzed module.
* UEFI Shell Script – was used for executing the fuzzed module.
* Custom made environment – C application that modifies memory according to the mutation and can be run under UEFI environment.
* Peach – Creates 100,000+ test cases (mutations).

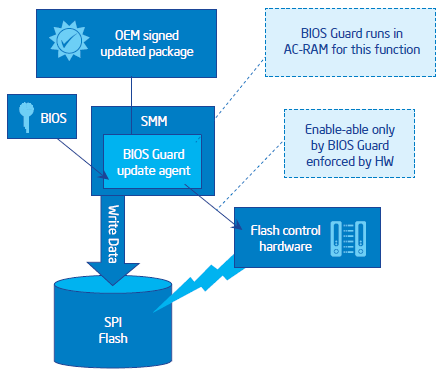
# BIOS GUARD AKA PFAT

A device's BIOS is contained in a privileged space that is invisible to anti-virus software. In addition, malware infecting BIOS remains persistent, and it does not go away even after a cold boot.

Thus its very likely that BIOS update vulnerabilities will continue to be exploited in the future. For this reason, platform security begins with BIOS.

Intel Platform Protection Technology with BIOS Guard, a new feature on 4th generation Intel Core processors (U-series), ensures that updates to system BIOS flash and Embedded Controller (EC) flash are secured. BIOS updates are cryptographically verified by the BIOS Guard module, using a protected agent running in AC-RAM (Authenticated Code RAM), to perform authentication and updating of flash control hardware.

For EC updates, BIOS Guard employs early BIOS POST provisioning of random secret that is shared between the CPU and the EC. The BIOS Guard module uses this value to identify itself to the EC, which will reject all protected operations without this identification.



**Figure 1. BIOS Guard**

## EXECUTION ENVIRONMENT

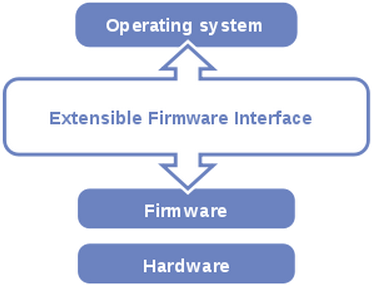
The environment was split into two sections. The first section is a monitoring PC which has Peach, ITP, UEFI Development Kit (EDK) and Python installed on it. This machine is a typical Windows 7 OS machine. All the pre-automation development tasks were developed and compiled using this machine.

The second machine is Intel development board with BIOS Guard enabled. The development board had not OS, only UEFI and its shell. On this machine we have executed the automated process which involved UEFI script, Test Framework (developed especially for BIOS Guard fuzzing task) and BIOS agent (few lines of code that were added to the original BIOS code) this helped the Test Framework to complete the whole picture.

The development board has no locks on the SPI flash thus the flash could be rewritten as much as supported by flash (what cannot be done on regular user PCs, since the flash gets locked after OEM puts its BIOS on it). Unlocked flash allows the developer to test the customized BIOS build (for example) for the development Intel board.

# What is UEFI?

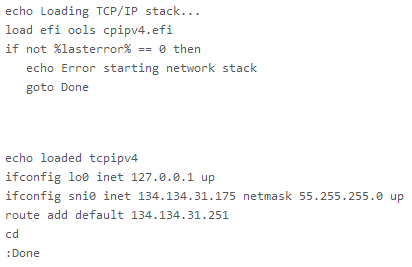
The Unified Extensible Firmware Interface (UEFI) is a specification that defines a software interface between an operating system and the platform firmware. UEFI is meant to replace the Basic Input/Output System (BIOS) firmware interface present in all IBM PC-compatible personal computers. In practice, most UEFI images provide legacy support for BIOS services. UEFI can support remote diagnostics and repair of computers, even without another operating system.



**Figure 2. UEFI**

BIOS Guard was executed from UEFI Shell environment which in some way reminds of an old DOS operating system. This Shell allows to run UEFI scripts and C application (Test Framework) that were compiled against UEFI Development Kit (EDK).

UEFI Shell allows simple scripting language that made it possible for a simple automation process to run in a loop until all test cases were carried out.



**Figure 3. Example of UEFI Shell Script**

# WHAT IS EDK?

The EDK is the open-source component of the "Framework", it is Intel's implementation of the EFI Specification, which was developed under a project codenamed "Tiano". The EDK was released under the BSD License.

The EDK is essentially a container for the Framework's Foundation code and sample drivers. The EDK is also a development kit for developing, debugging, and testing EFI and Framework drivers, EFI Option ROMs, and EFI Applications for use in the Framework environment.

# THE PROCESS

The process is devided into three subprocesses:

1. Creating test cases – Peach fuzzing tool created 100,000+ test cases for the next subprocess.
2. Executing the automation test environment - Applying the created test cases from the previous subprocess on the memory, executing BIOS Guard with these changes and collecting the response when BIOS Guard stops executing using UEFI Script, Test Framework and BIOS Agent.
3. After process #2 finished its job (after about three days), last process analyzes the data using Python script by comparing each test case to see if the response matches the test case.

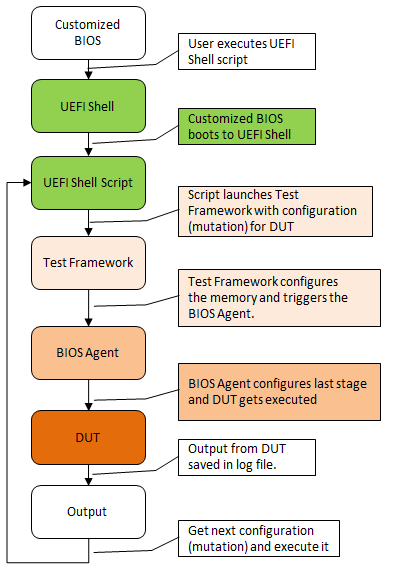
# Fuzzing approach.

For this particular task we chose to work with the Peach tool. Creating test cases using Peach is very flexible when penetration tester needs to characterize a protocol or any other data structure. BIOS Guard has its own data structure which is seperated into a couple of sections, so the main task of Peach was to create lots of test cases where each test case changed the structure from its original and valid form. The approach was not generational fuzzing because it would have created bilions of test cases when 90% of them are not needed, so the target was mutational fuzzing in which Peach takes a valid data structure form and creates mutated data form in specific places that were predefined by the designer of the Peach PIT file.

# Automated Environment.

This section was split into three main parts.

1. UEFI Shell script – The main iteration process.
2. Test Framework – Mediator between the script and BIOS Guard execution module.
3. BIOS Agent – The last process before BIOS Guard execution.



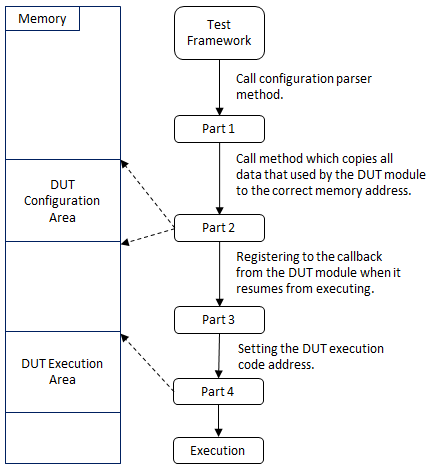
**Figure 4. Automated Test Flow**

Figure 4: Represents the simple flow of how the automated test was executed. The parts of Peach and data analysis by Python script are not in this flow.

# Test Framework

The mutated test cases forwarded from the Peach part should be passed to the BIOS Guard module. Each test case must be eventualy placed in the right memory location. Although UEFI script has the ability and access to most of the memory, we didn't choose script approach for this part. Instead we decided to create our own test framework which was built and compiled using EDK (UEFI Development Kit). We basically created an application running on top of UEFI. As metioned previously the Shell Script has access to memory, but it does not have other implementation that was needed for our task – EDK on the other hand does, for example libc capabilities.

The important part of this test framework is to fill the memory in a specific location with configuration data, execute the module and collect the response from BIOS Guard by registering it to the callback of the BIOS. Important to mention that the test framework cannot work on any version of BIOS. In order to execute the test framework certain amount of code was inserted into the BIOS, which means we specifically compiled the BIOS and burned it into the SPI flash chip where the BIOS is located.

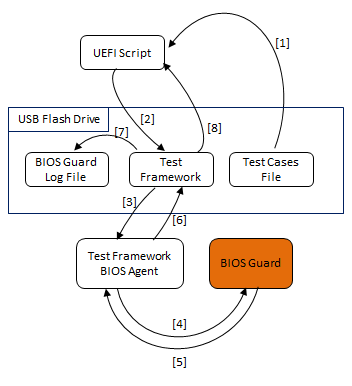


**Figure 5. Test Framework Flow**

# Analyzing The Results

BIOS Guard's response was saved in a log file. Log file contains row for each test case created by the Peach, in every row there is data such as error code if exists, return status and etc. It can be not very intuitive to analyze the results manually as the amount of test cases exeeds 100K. In order to analyze the results we used Python script. The script would iterate over the results, parse the data and compare the response to the request that was passed to the module. If the expectations weren't met the test case was documented for further investigation.

# The big picture of the process



**Figure 6. The Process**

Figure 6: Explains in more detail how the modified data is passed from one part to anoher till the end of the iteration.

**[1]** – UEFI script fetches the test case from a file located on a USB flash drive. (Each test case is a separate file)

**[2]** – UEFI script calls the Test Framework and passes the pointer to file location as an argument.

**[3]** – Test Framework parses the data located in the file and copies this data to relevant memory locations, after that it calls the BIOS agent.

**[4]** – BIOS agent constructs the environment for BIOS Guard execution and eventually executes it.

**[5]** – BIOS Guard signals that it has finished executing and saves the response to a register.

**[6]** – BIOS agent reads the response and executes the callback where Test Framework is registered.

**[7]** – Test Framework parses the response and creates user friendly response which is saved to the log file.

**[8]** – UEFI script is ready to fetch the next test case.

## METHODOLOGIES

Profound knowledge in the BIOS Guard module architecture was the main factor in creating a task plan. Without knowing the details of how the module works it would be hard to perform such a task.

Understanding how the environment itself executes the module is also important since without this knowledge it wouldn’t be possible to implement in the test framework.

Another important part is to understand every single aspect of the BIOS Guard data structure on which it relies during the execution. Not all parts were vulnerable, but must have been investigated.

In addition there was a need in understanding where would be the most relevant place in BIOS code to put the test framework agent. Without it we would have had a difficult time monitoring the response from the BIOS Guard.

Regarding the fuzzing part of the project it was important to understand what part of data architecture Peach should modify. As explained above, if the approach was “generation” the amount of test cases would have been immense and the time that would take to execute all the test cases would be over a month. So the approach was “mutation” that would start from valid data structure and change only the relevant parts of it, that way the amount of test cases was reduced to as few as 100K instead of billions test cases.

There were some other things we had to master in order to be able to pull this off:

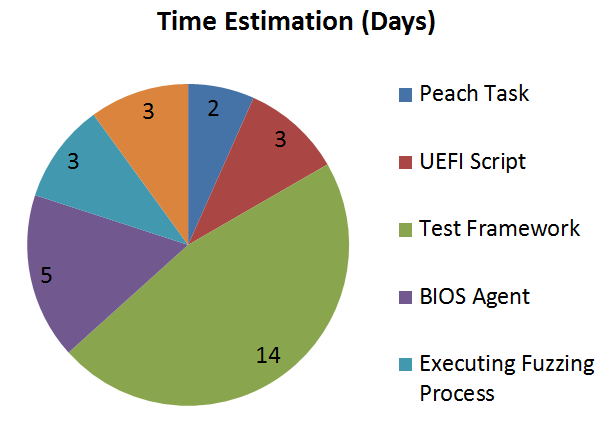
* Learning the UEFI SDK (EDK) and how to build and compile application using it.
* Knowing how to trick the system and access privileged memory addresses.
* One of the prior tasks was to learn the IA-32 architecture, knowing how and why BIOS Guard relies on it.
* Learning the UEFI environment and creating UEFI script which triggers the test framework and iterates over the created test cases.
* Use of Python to create a smart script which is able to verify if the response for each test case was correct. If not let the security tester know about it.

## CHALLENGES

There were quite a few challenges in this task:

* Establishing a working environment of the design board with correct BIOS version.
* Environment without Operating System made it tough. There was no friendly framework that could come handy during the task.
* Almost every tool that was running in the UEFI environment was created especially for the task’s purpose. Only pre and post automation tasks used commomn tools – Python and Peach.
* Developing the test framework was the biggest part in the fuzzing process, without it no fuzzing idea would work. Creating this tool was challenging for its dependency on BIOS environment. In order to test the compiled code we needed to copy the EFI file to USB flash drive and execute it on a machine with BIOS Guard technology, this consumed a lot of time.
* Creating a small modification to the original BIOS code. And again the most frustrating part of the the development and the only way to test the changes was to compile the BIOS source code with the changes then copy the BIOS code into the SPI flash.
* Deciding how the test case data will be passed to the Test Framework. The chosen approach is described in this paper, however there was another proposal which was not selected because of the challenge it presented for the implementation.

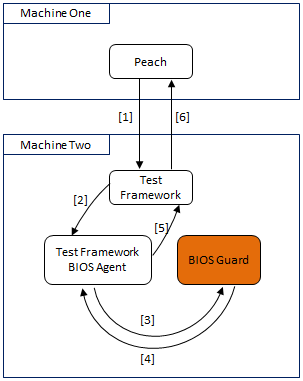
Time estimation and effort for this task are shown in the pie chart below.



**Figure 8. Time Estimation**

The initial idea was to use Ethernet card driver to communicate with other machine. One machine would run Peach which would create the test case, open socket to the second machine and send out the test case. On the second machine the Test Framework would open socket listener using UEFI Ethernet drivers and listen to the incoming socket, when the connection is created and the test case is received the Test Framework would execute the test flow.

Figure 7 illustrates the flow:



**Figure 7. The test process using Ethernet**

**[1]** – Peach creates test case and sends it to the Test Framework.

**[2]** – Test Framework parses the data received from Peach and copies the data to relevant memory locations, after that it calls the BIOS agent.

**[3]** – BIOS agent constructs the environment for BIOS Guard execution and eventually executes it.

**[4]** – BIOS Guard signals that it has finished executing and saves the response to a register.

**[5]** – BIOS agent reads the response and executes the callback where Test Framework is registered.

**[6]** – Test Framework parses the response and sends it back over the opened socket signaling to send the next test.

This idea is better and much more flexible with less stages in the middle of the procces.

The reason it was not used was because we weren’t able to make the Ethernet driver to work with on board Ethernet card.

# CONCLUSION

BIOS is located in privileged memory area and no anti-virus software can reach these areas. Attackers want to avoid being detected by anti-virus software thus BIOS is the perfect victim. Moreover if attacker succeeds to get the BIOS infected it gives him a powerfull foundation to insert malicious content before the Operating System is booted. If BIOS gets successfully infected the rootkit will stay forever unless SPI flash is rewritten with clean BIOS or the BIOS gets updated and the rootkit is patched (of course in case the rootkit is not smart enough to disable SPI flash update).

This paper gave the reader an overview of the work that’s been done for this task, gave two approaches that were considered in order to fuzz the BIOS environment (one of these approaches was used for the real project).

Hopefully it will give ideas of how to solve problems that a security architects are facing with similar tasks.

The fuzzing task is really important and it had a great impact on the whole security penetration test that was performed on BIOS Guard as this technology is already enabled in Haswell processors and has stepped into the market, for the integrity of Intel products is very crucial in nowadays competitive world.

The results of this task cannot be published for Intel security matters. The main purpose of this paper is to introduce the reader to fuzzing essentials and share this knowledge among other security researches.

# ACKNOWLEDGMENTS

# I would like to acknowledge our co-workers from SSG eMMDi especially Cohen Nehemia, Eli who has developed the Test Framework.

# REFERENCES

[1] – White Paper, Intel Hardware-based Security Technologies for Intelligent Retail Devices, <http://www.intel.com.tw/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf>.

[2] – Peach Fuzzing Platform, <http://peachfuzzer.com/>

[3] – UEFI Development Kit, <http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK>

[4] – UEFI Shell, <http://software.intel.com/en-us/articles/uefi-shell>

# AUTHORS’ BIOGRAPHIES

 Yevgeniy Kiveisha ([yevgeniy.kiveisha@intel.com](mailto:yevgeniy.kiveisha@intel.com)) is a software engineer in Intel’s IT Flex Services for the last 2 years.

Graduated B.Sc in Electronic Engineering.

Experience in .Net, C#, C++, JavaScript, Embedded Environment.