Analysis of Security Possibilities of Platforms for 3D Graphics

Tomas Svoboda, Josef Horalek
University of Pardubice, Faculty of Electrical Engineering and Informatics,
Pardubice, Czech Republic.
tomas.svoboda5@student.upce.cz

***Abstract*— This paper introduces the results of an analysis of security threats based on 3D graphics interface abuse mainly in Windows operating systems. Basic security threats, which can greatly influence the functionality and security of an operating system, are introduced in this paper. Due to the specific nature of the operating systems architecture, the most vulnerable part are the third party drivers, which have access to the core mode. Different approaches of protecting drivers are described together with disadvantages of these solutions. Practical methods of attacks are discussed in detail and when relevant, program alterations are suggested to program designers. The goal of these suggestion is to minimize or to completely eliminate mentioned security threats, making 3D graphics interface more secure.**

***Index Terms*— 3D Graphics; Security threats; Third party drivers.**

I. INTRODUCTION

Thanks to IT specialist paying extra attention to security, usual pathways for malicious code have lately been either fully closed or very difficult to discover. Standard security threats are the topic of many defense strategies and approaches, such as layout randomization, utilizing NX security, rights restrictions within the operating system, locking authorization, utilizing isolation on application level in the form of integrity level or in the form of utilizing a sandbox with operating system interface restrictions.

For this reason, the focus of attackers shifts to other parts of the operating system, through which the attackers could break through the security measures and which are hard to secure at all. Generally, we are speaking about any random code in core mode, which can use any processor instruction and thus access any sources. Code of core mode can be divided into the core itself, drivers, and system services. The only part, in which the highest percentage of third party codes is present, are the drivers needed to ensure the functionality of hardware or specialized software and thus the highest probability of critical errors is to be expected. The fundamental problem of drivers lies in the fact that the outer code is divided from them by thin interface oriented on power and not prepared to be used as a security divider. An example of such drivers is graphic card drivers, libraries, and services associated with them. Although the part of core interface is accessible only as system calls, graphic cards and their drivers are a potential security threat.

Standard drivers are hidden from potentially untrustworthy codes using several different interfaces, parameters and data are processed and validated by the system. Specifically altered driver using DMA channels providing direct access to the memory for reading and writing directly into physical memory is far more common than an attack aimed at general drivers. Such process can be both, an attack or a rootkit detection method.

Unlike standard drivers, graphic drivers mostly utilize separate interface and they interpret and perform complete code given by a client application. The result of this is a significantly better visible area for an attacker, and because WebGL standard, enabling Javascript access to graphic interfaces in webpages, exists, potential attack does not require anything else than malicious code attached to a webpage, e.g. in the form of a compiled script.

The aim of the paper is to investigate possible pathways to breeching system security and to suggest security methods which could be applied to graphic card drivers.

II. DEFINITION OF PROBLEMS AND THEIR ANALYSIS

Recently, several approaches to how to deal with security threats of general drivers exist, but almost all of them are dependent on using virtualization of the first type and a hypervisor as an isolated environment for monitoring and checking the drivers.

First option, which is not using virtualization, is an approach described in [1], in which an adaptation of code instrumentation using a translation of binary code is suggested. This approach is primarily designed for operating systems based on a Linux core, which differs greatly from Windows in its implementation of core mode and services. Another problem is that this approach can be applied on a GPU code, as it is processed solely on a driver and afterwards by hardware and it is therefore not possible to prevent interference through the GPU. The last issue with this approach is the power demand of the graphic card and its influence on its performance.

Next solution is utilizing full virtualization for DriverGuard (DG), as introduced in [2], in which one can use a lite version of a hypervisor to put the control code above the code of the controlled system, while the supervisor also has to be able to distinguish individual types of allocated memory so it is possible to distinguish the program part of memory from the data part. The principle lies in monitoring changes in the data area of the memory of the driver, so when DG accesses a page

of a secured area, it forces throwing an exception, resulting in stopping the code execution. Just like when using classic debug function, int 3, without the driver or the malicious code could recognize such action. DG then analyses the source of the access and based on the analysis, it either permits or denies access.

Another group of solutions for this issue is virtualization of only selected components and drivers, as introduced in [3] as SILVER. Each object in core mode is encapsulated in its own virtualization with a common hypervisor. When accessing the particular objects or executing their code, the impossibility of direct modification of paging tables is used and instead, the tables itself are being switched between, and thus only a restricted view of the core memory with optional different permission for each object is possible.

The solution introduced in [4] utilizes a hypervisor for monitoring writings into secured areas in memory and also forces the memory department with sensitive data structures from others, because it is not possible to efficiently use smaller allocation unit than the memory page, which is minimally 4kB on x86 processors, which would not permit using the assistance of memory unit of the processor. The first problem is the dependency on virtualization, which might not be available all the time, and without virtualization of I/O ports, it is not possible to deny the GPU access to secured memory efficiently. Another problem is the impossibility to secure data structures of drivers, mostly data drivers of the graphic card.

The issue with abovementioned solutions is the non-zero effect on system performance and in case of graphic drivers; those solutions cannot be suitable for majority of users who demand the highest GPU performance.

Besides the effect on the performance, possible incompatibilities with given hardware and the way of its utilization are an issue for general hypervisors, as the operation system itself has high requirements on virtualization.

Virtualization security, such as SILVER, requires larger disadvantageous modification of the system, in order to execute necessary allocations of core objects and their allocation to individual virtualizations. A significant disadvantage is then noticeable effect on system performance, as it is necessary to have extensive code for memory administration, and each switch between objects, or between the user and the core modes, will be significantly more difficult.

For all abovementioned solutions, there is one characteristic common disadvantage, because they require the system to support virtualization extension. For x86 an x86-64, exist two pairs of virtualization technologies. VT-X and VT-D by Intel and AMD-V and AMD-Vi by AMD. VT-X and AMD-V extensions provide basic support for virtualization and administration of a hypervisor, host system, and guest systems. The support consists of instructions administrating virtualized systems and possible extensions of system objects, such as extended page table (EPT) or advanced programmable interrupt controller virtualization (APICV). VT-D and AMD-Vi are virtualization extensions that extend basic virtualization by support for virtualization of input-output sources. The aim is extending the isolation of the guest system from the viewpoint of working with hardware without noticeably affecting system performance, which would be cause by software emulation and hardware translation. These extensions offer the option to utilize memory addresses designated for direct memory access (DMA), routing interruptions, and supporting direct sending of interruptions from the virtualized devices to virtual processors.

Another solution is DeviceGuard. It is a function implemented in the core of Windows 10 operating system, which uses virtualization to protect processes working with signatures, system processes and drivers of devices, from outer attacks. The disadvantage of the whole system are system requirements, as it is necessary to ensure correct implementation of UEFI firmware with the SecureBoot function, which verifies the integrity of booting up the system and Hyper-V virtualization, which requires both extensions, the VT-X and VT-D.

In Windows 10, there are also Shielded VMs, running under Hyper-V, which contain restricted SKernel, which does not support any drivers, except for necessary minimum, and which enables to host sensitive system services, such as Local Security Authority (LSA) in such a way, that sensitive information contained in them are secured. It is mostly the security of logging information against an attack from an administrator account.

In present days there is no solution that would deal with the issues of the security of graphic drivers and work with GPU without utilizing virtualization, negatives of which have been mentioned above. Most of the current work is focused on optimization of 3D rendering in mobile devices with no focus on enhancing security [5,6].

III. Security Threats and Their Solutions

Among the main analyzed security threats and vulnerabilities belong Buffer over-flow, Timing attack, Object slicing, reading and underflow of character buffers, invalid indicator, Use-after-free, Out-of-memory, and Malformed input data. Each security threat was assessed based on its significance and the possibility of redress.

A. Buffer Overflow

It is a standard error, during which the code writes behind the border defined as the area of the memory. This way, the attacker may in some cases rewrite the data structure, in which the given buffer is located.

The main aim when abusing this error, is rewriting indicators in such a way that they would link to a memory under the attacker's influence, or which contains content provided by the attacker and with a known address. The content is then a machine code, which can attempt to abuse other vulnerabilities to gain higher permissions or to execute target activity. This flaw and this particular way of abuse for user code throws an exception, as the access memory is solely in pages with prohibited code execution. This restriction is present since Windows XP Service Pack 2 and it primary uses one of the free bits in the description of a page labeling restrictions. The processor supporting this function throws a special exception, which then terminates the process.

This measure cannot be applied by default drivers in the

core mode, as an exception will cause fast system termination, because at that moment, there is no guarantee for any non-standard exception that structures and allocated memory for the core mode are not damaged. The drivers thus must explicitly call for such labelled memory and this option is available since Windows 8.

Attack alternative to the previous one is the rewrite of the NULL ending character by a random byte, and abuse of incorrect work with standard functions for working with strings in C language, which use the NULL character to determine the end of the string. In such case, the error cause the possibility to read the memory behind the character string and it enables possible data leak or getting information necessary for subsequent attacks. Alternatively, the memory can be rewritten by the following code and thus cause a buffer overflow with the safe effect:

```
Void      VulnerableFunction      (wchar_t
*buffer,int cbLength)
{
char* dest = malloc(…);
memcpy(dest,buffer,cbLength*sizeof(wchar_
t));
…
}
```

B. Timing Attack

It is a class of errors in code, when the function does not validate the input from the calling code correctly, or it validates it prematurely, and the attacker then can, thanks to more core processes, change parameters from a different thread and bypass authentication code, as depicted on Figure 1.

The basic assumption is that the function takes over one of the input parameter, object indicator, basic parameters of which can be altered parallel in a different thread. The aim of the attack is to cause buffer overflow or another error, e.g. a free indicator. The most significant disadvantage of such an attack is an inherent dependency on a programming error, when the access to a given object from more threads is not treated correctly and race conditioned, and depending on many factors, such as sequencing threads for performance or system activity of the core, the attacker's thread can but does not have to have the window to parallel manipulation with an object.

Standard security for such kind of attacks is validating borders of array for every iteration or disablement of the internal structure of an object for an outer code and forcing access through methods, which force synchronization through threads or which fill in missing synchronizations.

The disadvantage of such defense is the necessity to validate the buffer for every iteration. Because modern processors possess high efficiency of branching predictor, but that means at least several instructions more. In addition, such solution will prevent any code vectorization, i.e. utilization of specialized instructions for parallel processing of several elements at once. In the case of AVX extension for instructional set x86/x64, the number of elements can be as high as 8 elements for one instruction. In other cases, the condition requires performing functions that are more complex and thus reaching further loss of performance.

Locking the memory page belongs among the suggested methods of security dealing with timing attacks. With buffers it is possible to force allocation in individual pages and it is also possible to change page security before validation from read/write to "read only" using the VirtualProtect function, which instructs the memory administrator to make changes to memory attributes concerning security of pages containing the given extent defined by an address and size. Subsequently, a copy from memory allocated by the user is made into the memory controlled by runtime or a driver. If it is not possible to ensure the use of an allocator, which uses separate pages for allocation, it is then possible to introduce change of memory administration, which will enable to lock the heap, in which the given array is located an thus prevent any changes.

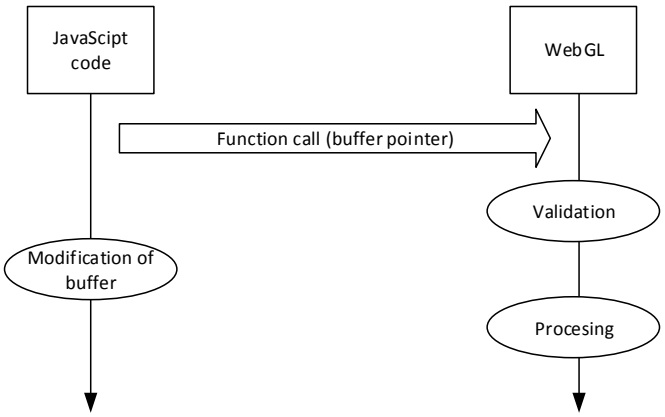

Figure 1: Timing attack

Another possible solution is strict access synchronization. With objects of graphic interface, it is possible to rather easily ensure complete access synchronization. To achieve scalability, it is possible to utilize Slim Reader/Writer (SRW) stamps, which were located in individual public methods. An alternative to access synchronization is attribute locking, i.e. introducing a type bool variable, which will deactivate codes of other methods for the given instance of the object after calling an executive method.

We recommend to switch from C code to a C++ code and to utilize STL (standard template library), which allows to use iterators to check the extent.

C. Object Slicing

This security issue occurs primarily in object-oriented languages, in which a class can extend the definition of a superior class by another attribute and then during transferring the object to a function, which expects a superior class, the object can have the added attributes trimmed. With indicators, the trim itself is not realized, but methods which will be called and which will belong to the superior class, will not know about other attributes. If one or more methods were deployed within an inferior class, they will behave according to the new code and subsequently perform incorrect functions, or they will lead to a security flaw.

The same problem applies to the procedural C language, in which the code can define several structures with a common base and of similar structure. The design of the solution to this issue lies in two complementary approaches. In the first case,

additional variable, which determines the version of the object, is introduced in classes. If the variable is introduced at the beginning of the declaration, it will be present even in case of trimming. Another method is using RunTime Type Information of the given language and in the event of transferring indicators, it is possible to verify their relevance.

D. Character Buffer Overflow and Underflow

Character buffer overflow and underflow is a combination of a buffer overflow with a timing attack when character buffers are used for functions that process files or shader functions for saving the name of the file of a shader, and these buffers are often controlled by a vulnerable code, or directly by a code trying to penetrate it. Vulnerable code can be Javascript, which runs under the web browser or in independent environment, such as V8, and which is used as a 3D interface of WebGL. Because initial validation of strings is performed during transferring names from the user code into the WebGL interface and then to components responsible for processing requests, there is an interval between the initial validation and the finish of copying into the buffer. Such buffer is controlled by a responsible component. Penetrating code, which is running within the original Javascript and which has access to the source string, can then attempt to change the content to a shorter or longer string which contains malicious code or other data specific for the object containing the target buffer. If the code is loaded within this interval, the vulnerable function then rewrites the content not only of the target buffer, but also the rest remaining part of the object.

E. Invalid Indicator

Invalid indicators are an important calls of errors, significance of which ranges from the option to cause an application or a service to crash, i.e. a DOS type attack, to forcing a run of a malicious code. The simplest type of an invalid indicator is a zero indicator, which links to a memory on the 0 address. This memory is not labelled as invalid on common platforms (IA 32/IA 64 or ARM). In order to detect common errors in programs, first 64kB of virtual memory are labelled as inaccessible and thus when attempting to dereference the zero indicator, an exception is thrown by a memory controller of the processor. Such exception typically terminates the process of the code does not filter this exception and does not attempt to continue. The most common reason for the existence of a zero indicator of the failure of memory allocation, or the attempt to extend the allocated array fail, as shown in the following example:

```
void          BadFunctionWithResizing(char*
string)
{
….
string = Realloc(string,new_size);
  ….
}
```

Vulnerable code rewrites the original indicator with a new one, which is returned by the standard *realloc* function. The problem is that if allocation of new memory fails, a zero indicator is returned and an error indicator and the original allocation are kept. However, in that moment the indicator of the original memory is lost and thus a memory leak is caused, but when attempting to use such indicator, an exception is thrown and the application crashes.

The following demonstration shows one possibility how to repair this security flaw:

```
void     CorrectFunctionWithResizing(char*
string)
{
….
char *string2 = Realloc(string,new_size);
if(!string2){return;}      /*or      throw
exception*/
string = string2;
  ….
}
```

The second class of invalid indicators is non-zero indicators. One of the sources of such indicators are variables, which were not initialized on the correct, or rather known, value like the zero indicators. Another cause is handling the value of indicators incorrectly (e.g. indicator arithmetic without proper control of interventions). This security flaw is more dangerous, as the indicator does not have to link into virtual space not accessible for the application (it is not allocated or it belongs into the core mode) and in case an attack code can ensure allocation in target address space or somehow affect the indicator, it is then possible to ensure the performance of a malicious code with rights of the given process or a service.

F. Use-after-free

Use-after-free is one option of an invalid indicator, in which the base of this error is vacating an object and subsequent deallocation from memory, and afterwards an attempt to call a method of this object or an access to a variable in it happens, while the memory previously occupied by the object is already allocated to a new object or it is being used by the still existing object. Such error often occurs when of the variables linking to this object does not reset to zero or when a new indicator was not saved for some reason. The result is then often crash of the code, as there is no executable code present in that location at that moment or the function execution is started on a random place. It is cause by not executing the prologue of a function, when the environment including the buffer settings for the given function is set. In less frequent cases, the whole function including the prologue is being executed, but when attempting to access the input parameters, random values, or zeros if the page was set to zero by the system, will be loaded and a failure occurs. Especially with a C++ object, in which the first parameter includes the indicator of the beginning of an object, then a zero indicator is not valid and no code is ready for such indicator. In both cases, the application fails and denial-of-service occurs. However, if the memory contains data from the penetration code, the content is executed in the given user context and for the components running with higher rights, e.g. drivers, it is a successful privilege escalation.

This flaw can easily be abused in Javascript, as it is a significantly asynchronous code execution, including simple explicit parallelization.

G. Out-of-memory

This vulnerability belongs among less serious flaws, as it cannot force execution of another code in most cases. It is a significant memory leak in the GPU driver, which is not administering the memory used for graphic sources correctly, and for module in core mode, it primarily concerns dynamic buffers used for allocation and repair pages for data intended to be loaded to the GPU. Although the memory for buffers in core mode should be labeled as pageable, it is not a required feature and a vulnerable driver can cause exhaustion of unpageable memory under allocation request of the malicious code and thus cause the instability and rare failures of drivers and services using said memory.

H. Malformed input data

In the context of interface for 3D graphics, data input for pipeline is one of the most important inputs. Each buffer containing data of peaks must be in accordance with pre-specified format, which defines the order of variables, their type and size, and sometimes how they are used. Main deviations are often determined by an initial validation, executed by the runtime of the graphic interface, and their processing is terminated. However, some anomalies cannot be revealed this way. It is mostly incorrect values (e.g. abnormal FP) or structures declared higher than in the buffer. Alternative attack is then the timing attack, when the interval between compilation of input and transfer of data to the driver for processing is used. An attempt to process data behind the border of the buffer and subsequent rewrite of internal structures can occur in the driver itself, which is similar to using buffer overflow. The extent of such flaw can range from DOS to executing a code under the rights of the driver.

IV. CONCLUSION

The investigated area of 3D graphic interface and the issue of it being attacked from the outside is very important topic in operating systems' security. We based our investigation on the principles of WDDM driver architecture and DirectX interface, which uses this architecture. It was discovered that majority of functions and services within DirectX is implemented in drivers from graphic cards. Therefore, the main burden, from the viewpoint of implementing security measures, lies on their programmers. Another important finding is the extent of threat of important functions for working with the pipeline. These functions use user defined data or various buffers, including basic strings, for access.

Various methods of penetration, which could be used against the interfaces and drivers implementing these interfaces were analyzed. Several various penetrations were identified and possible countermeasures appropriate for graphic interfaces were suggested. The significance of this paper is in its focus on identifying new methods of penetration into the system via various graphic interfaces and in suggesting of possible countermeasures considering their implementation on graphic cards, which can help programmers eliminate security threats.

ACKNOWLEDGMENT

This work and contribution is supported by the project of the student grant competition of the University of Pardubice, Faculty of Electrical Engineering and Informatics, Intelligent Smart Grid networks protection system, using software-defined networks, no. SGS_2016_016.

REFERENCES

[1] Feiner, Peter, Angela Demke Brown and Ashvin Goel, "Light-weight kernel instrumentation framework using dynamic binary translation," *The Journal of supercomputing*, 2013. ISBN: 0920-8542..

[2] Suzaki, Kuniyasu, Toshiki Yagi, Kazukuni Kobara and Toshiaki Ishiyama, "Kernel Memory Protection by an Insertable Hypervisor Which Has VM Introspection and Stealth Breakpoints," *Advances in Information and Computer Security*, pp. 48, 2014. DOI: 10.1007/978-3-319-09843-2_4.

[3] Xiong and Peng Liu. Silver, "Fine-Grained and Transparent Protection Domain Primitives in Commodity OS Kernel," *Research in Attacks, Intrusions, and Defenses*, pp. 103, 2013. DOI: 10.1007/978-3-642-41284-4_6.

[4] Srivastava, Abhinav and Jonathon Giffin, "Efficient protection of kernel data structures via object partitioning," *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 429, 2012. DOI: 10.1145/2420950.2421012. ISBN 9781450313124.

[5] Marek, T., Krejcar, O., "Optimization of 3D Rendering by Simplification of Complicated Scene for Mobile Clients of Web Systems," *7th International Conference, ICCCI*, Lecture Notes in Computer Science, vol. 9330, pp 3-12, 2015. DOI: 10.1007/978-3-319-24306-1_1

[6] Marek, T., Krejcar, O., "Optimization of 3D Rendering in Mobile Devices," *The 12th International Conference on Mobile Web and Intelligent Information Systems*, Lecture Notes in Computer Science, vol. 9228, pp. 37-48, 2015. DOI: 10.1007/978-3-319-23144-0_4.