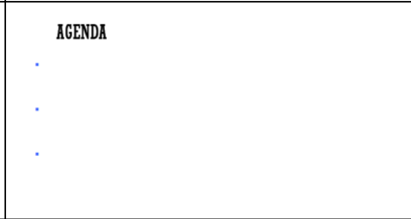
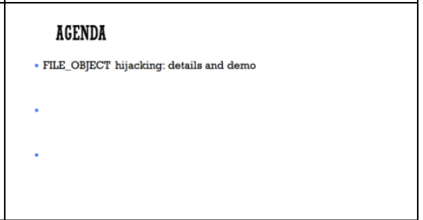
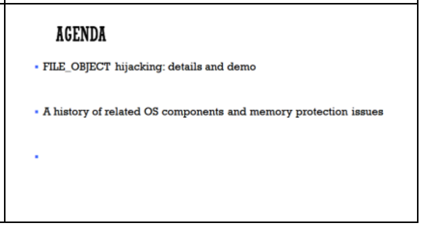
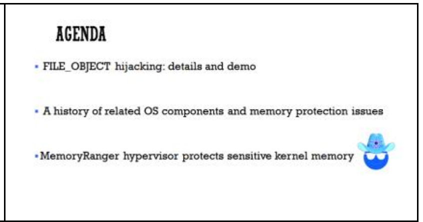
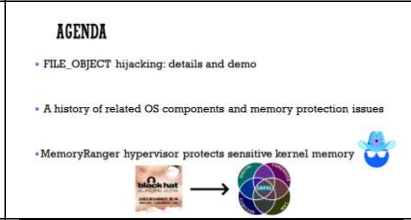
MemoryRanger Prevents Hijacking FILE_OBJECT Structures in Windows Kernel

Slide 1 Hello Hi! I’m Igor! Thanks for coming! Today I’ll present you how some Windows components such as the Security Reference Monitor and the Object Manager can be attacked and how to prevent these kinds of attacks using my MemoryRanger.	
Slide 2 “WhoamI” That’s my 6-th time here at the ADFSL conference and I’m very happy to be in Florida again. In my research I am focused on operating system security and most of **/ov/** all on Windows Kernel Security. You can find my results in my blog. Also, I am fond of travelling and powerlifting as well.	
Slide 3 “Agenda” Our talk will have three parts.	
Slide 4 “Agenda” Firstly, I’ll show you how to áccess the content of file, which has been opened without sharing access. This new attack on kernel memory does not trigger any protection mechanisms.	
Slide 5 “Agenda” Secondly, I’ll present you a brief history of memory protection issues.	

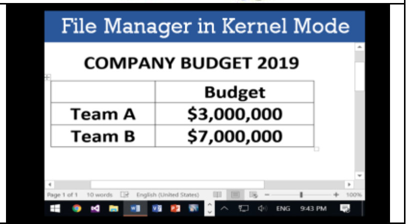
Slide 6 “Agenda”
Finally, I am going to demonstrate how this problem can be solved by my MemoryRanger.
MemoryRanger is a hypervisor /ˈ**haɪpə(r)**ˌ **vaɪzə(r)/** , which protects memory by running drivers inside the isolated /ˈ**aɪsə**ˌ**leɪtɪd/** kernel spaces.



Slide 7 “Agenda”
Last year I present<u>ed</u> my MemoryRanger at the Black Hat Europe.
Today’s presentation demonstrates its <u>further</u> development.
Feel <u>frée</u> to ask me any questions at the end of our talk.



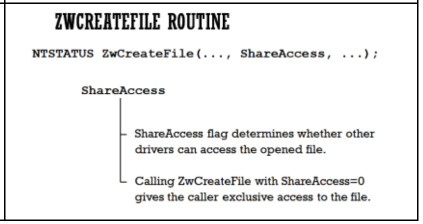
Slide 8 “The Situation”
Let’s imagine the following situation.
The big boss is sitting in front **/frʌnt/** of his computer and is drawing up a budget.
We can see it on the screen.



Slide 9 “The Situation”
Of course, the boss <u>doesn’t want to share</u> the budget with anyone and so he opens the budget file without sharing access.

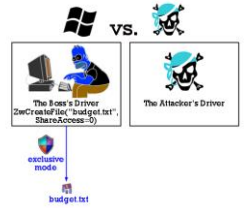
Slide 10 “The Situation”
To create such a file in kernel mode he uses function ZwCreateFile.
It has 10 parameters, one of them is ShareAccess,

Slide 11 “The Situation”
which <u>determines</u> whether other drivers can read or modify this file.
Drivers usually set ShareAccess to zero, which gives the user <u>exclusive access</u> to the open file.

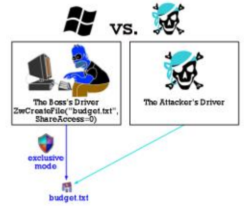


Slide 12 “The Situation”
Here is the “hacker-attacker” plotting to snoop **/snu:p/** (I mean read **/ri:d/**) and modify the boss’s budget.

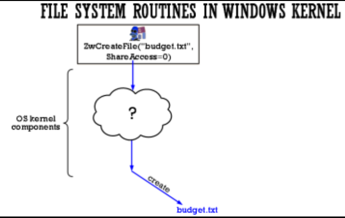
Slide 13 “The Situation”
Mícrosóft guarantees that the user gets an exclusive access to the open file and all other access attempts will fail.



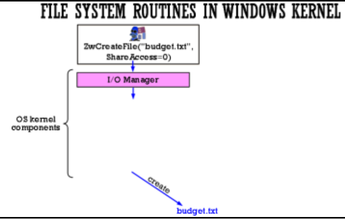
Slide 14 “The Situation”
However, the attacker can overcome this barrier and gain the access to the budget. I’ll show you how.



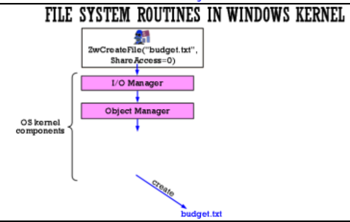
Slide 15 “Access the File by Hijacking its FILE_OBJECT”
Let’s have a look at the details of file system routines in Windows kernel.
Drivers use ZwCreateFile routine to create or open a file on the disk.
What happens after the driver calls this routine?

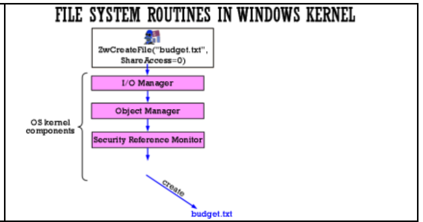
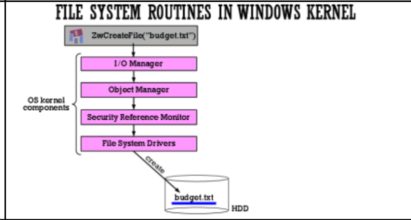
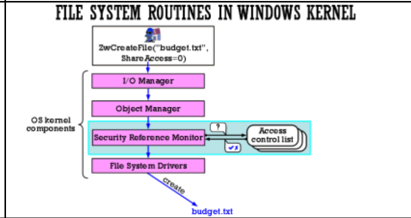
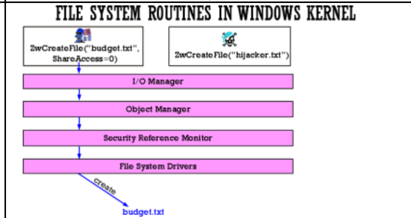
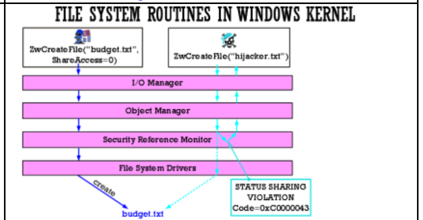
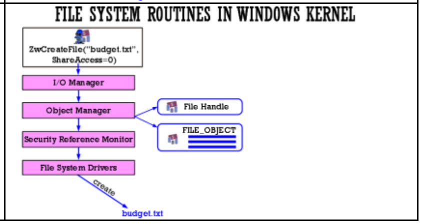


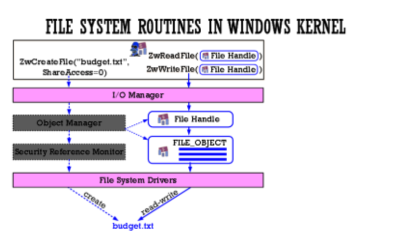
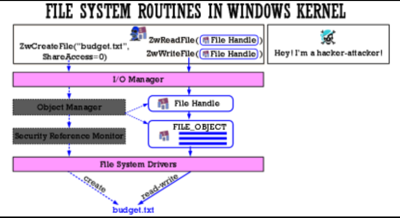
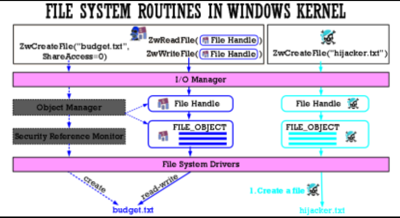
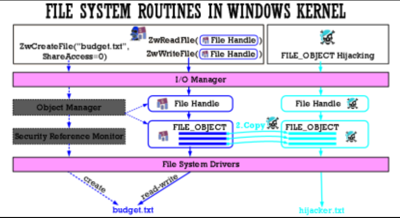
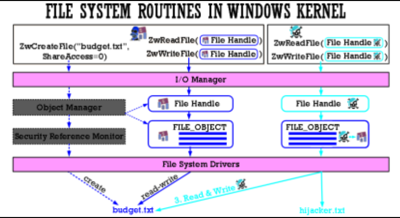
Slide 16 “Access the File by Hijacking its FILE_OBJECT”
This function involves an Iee/Oou Manager,



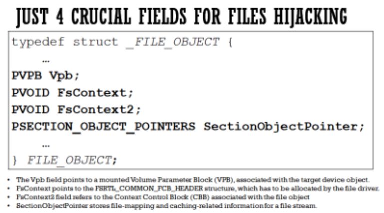
Slide 17 “Access the File by Hijacking its FILE_OBJECT”
Object Manager



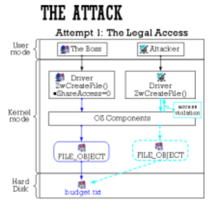
Slide 18 “Access the File by Hijacking its FILE_OBJECT” Security Reference Monitor	
Slide 19 “Access the File by Hijacking its FILE_OBJECT” and then the control goes to the file system drivers that create the file on the disk. The most important component is the Security Reference Monitor,	
Slide 20 “Access the File by Hijacking its FILE_OBJECT” which is in charge of the access validation. It processes the files request and gains or blocks an access to the file. Security Reference Monitor collects and checks file permissions using its Access Control List.	
Slide 21 “Access the File by Hijacking its FILE_OBJECT” For example, if the file is created without sharing access or in an exclusive mode.	
Slide 22 “Access the File by Hijacking its FILE_OBJECT” All other access attempts to open this file fail {pause} because they cause the conflict with the sharing mode of an existing file handle. In that situation Security Reference Monitor returns the status **/ˈsteɪtəs/** sharing violation. ZwCreateFile routine always involves Security Reference Monitor, which prevents illegal access to the files, that are opened at the moment.	
Slide 23 “Access the File by Hijacking its FILE_OBJECT” Well. After Security Reference Monitor allows **/əˈlaʊz/** access, the Object Manager creates {pause} a fíle hándle and a FILE_ÓBJECT. That FILE_OBJECT structure is the internal análógue **/ˈænəˌlɒg/** of the file handle, which is used as a parameter in read and write functions.	

Slide 24 “Access the File by Hijacking its FILE_OBJECT” The functions ZwReadFile and ZwWriteFile access the file using the already created FILE_OBJECT structure. The key point is that they do not involve any security checks. Read and write file operations are processing without calling Security Reference Monitor. The attacker can use this (so-called) zero-day vulnerability **/ˌvʌln(ə)rəˈbɪləti/** to compromise the open budget file in the following way.	
Slide 25 “Access the File by Hijacking its FILE_OBJECT” Here is the hypothetical attacker.	
Slide 26 “Access the File by Hijacking its FILE_OBJECT” Firstly, he can create a new file using ZwCreateFile. We call it the “file hijacker”. OS returns the corresponding file handle to the attacker. Next attacker can locate its FILE_OBJECT structure and the budget FILE_OBJECT structure, which is related to the budget file.	
Slide 27 “Access the File by Hijacking its FILE_OBJECT” Secondly, the attacker can copy the budget FILE_OBJECT to the FILE_OBJECT of the file hijacker. Let me call it the hijacking attack.	
Slide 28 “Access the File by Hijacking its FILE_OBJECT” And now all read and write operations to the file hijacker are redirected to the budget file. As a result, using the file hijacker handle, the attacker will be able to read and modify the budget file using functions ZwReadFile and ZwWriteFile. Now let’s move on to the FILE_OBJECT details.	

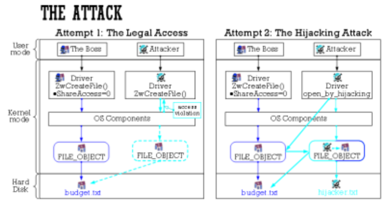
<table>
<tr><td>

Slide 29 "Access the File by Hijacking its FILE_OBJECT"
The FILE_OBJECT is a kernel-mode structure that represents an open file.
It includes 30 thirty data fields and is documented in the Windows Drivers Kit.
In my experience drivers usually do not read or modify the fields of FILE_OBJECT structure.
But in this research, I <u>did read</u> and <u>did modify</u> those fields.
The thing is that just by copying the following four data fields from the budget FILE_OBJECT the attacker can read and modify the budget file. Let's see how it can happen.
</td><td>


</td></tr>
<tr><td>

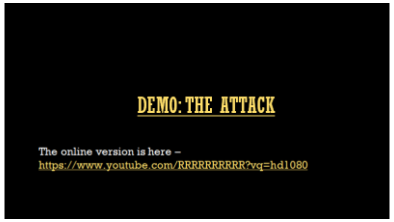
Slide 30 "Demo: The Attack Scenario"
I'll show the following scenario.
The boss will create a budget file in an exclusive mode and will draw up the budget.
After that, the attacker's driver will try to access the budget file. The attacker will try to access the budget file twice. First, by calling ZwCreateFile routine
</td><td>


</td></tr>
<tr><td>

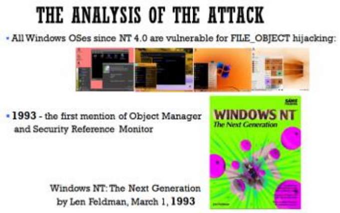
Slide 31 "Demo: The Attack Scenario"
and second by using the proposed hijacking attack.
To manipulate **/məˈnɪpjʊleɪt/** with files from kernel-mode I will use drivers.
The input data will be sent to these drivers using the corresponding user-mode console apps. Let's see!
</td><td>


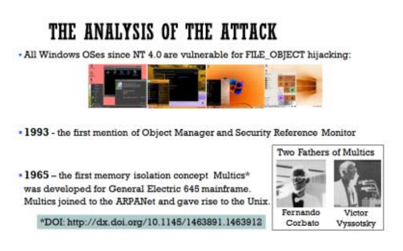
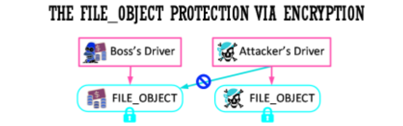
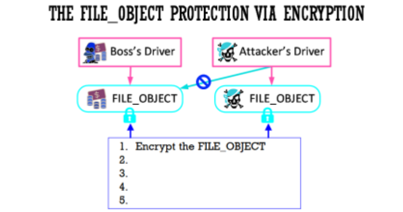
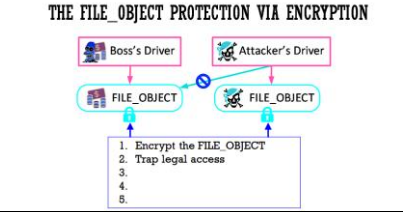
</td></tr>
<tr><td>

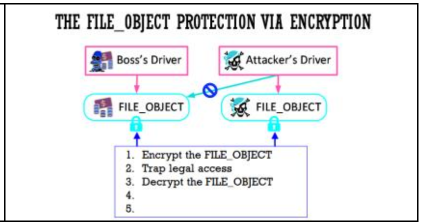
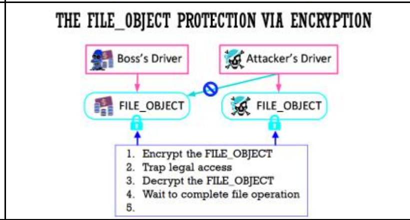
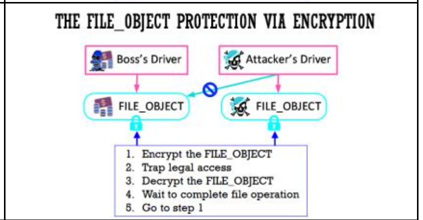
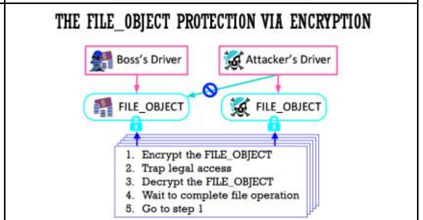
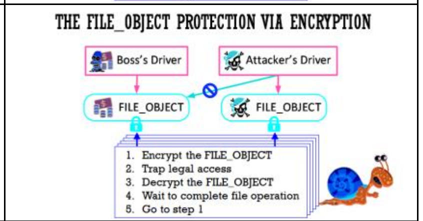
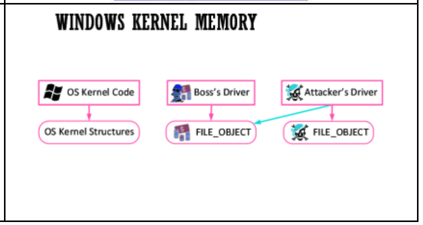
Slide 32 "(Demo: The Attack)"
The boss is launching his console, which loads a driver.
By using **f_open** he creates an empty budget file in the current folder.
Windows creates the file handle and allocates the FILE_OBJECT structure.
By using **f_write** the boss is setting up the budget.
Now, the boss is checking the budget. Done.
The boss is talking on the phone and the budget file remains open.

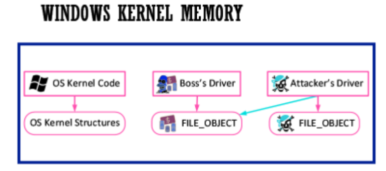
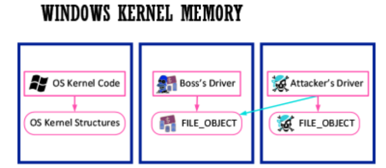
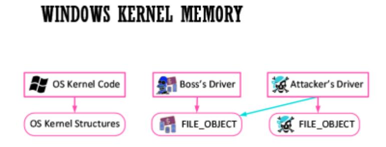
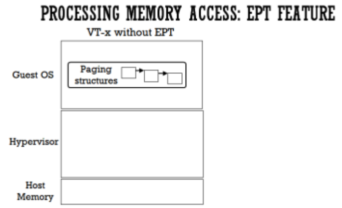
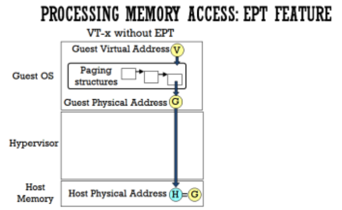
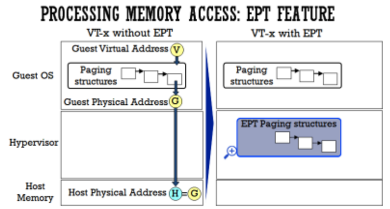
At this moment, the attacker has got a chance to snoop the budget.
He is launching its console. He is calling file system routine ZwCreateFile to open the budget.
So, he fails to open the file. Windows returns the status sharing violation.
Windows prevents illegal access to the budget file.
</td><td>


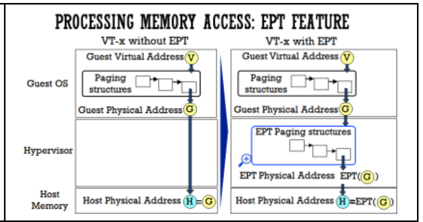
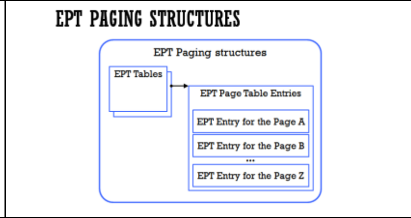
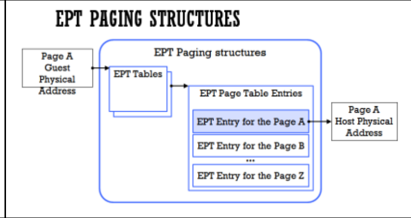
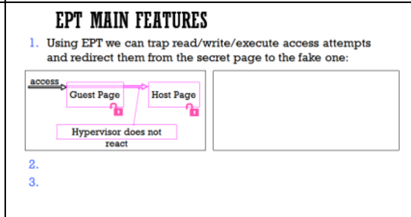
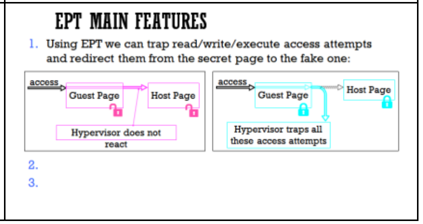
</td></tr>
</table>

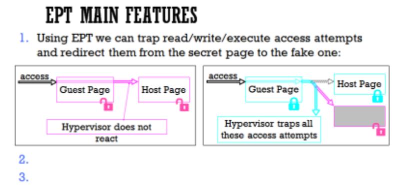
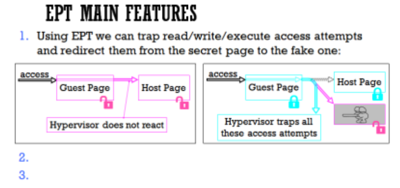
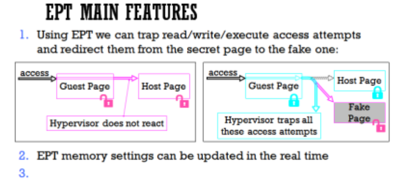
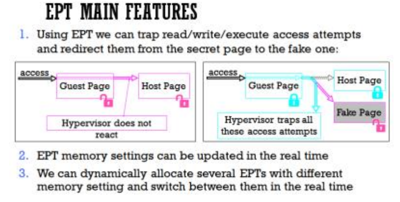
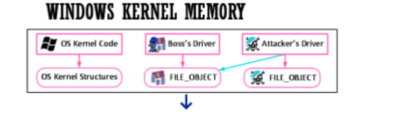
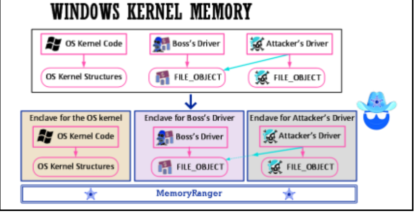
But the attacker keeps trying to gain the access. He is copying the address of the FILE_OBJECT to run the hijacking attack. By using **f_open_by_hijacking** the attacker creates a file hijacker. Windows returns the corresponding file handle and FILE_OBJECT. After that the attacker copies the budget FILE_OBJECT into the allocated FILE_OBJECT structure. Now, the attacker is reading the file hijacker. And we can see that the budget is revealed, it is not a secret anymore. The attacker is coping the content in order to modify it. The attacker is overwriting the budget file by replacing the data. And now if the boss decides to check his budget, he will see some illegal changes. But let's wait for PatchGuard reaction, which is designed to prevent such illegal memory modifications. Usually PatchGuard crashes the OS in less than one hour. We've been waiting for 8 hours it is quite a long time but nothing has happened. The OS has not been crushed. It means that PatchGuard fails and the OS is unable to guarantee an exclusive access. The system is not protected.	
Slide 33 "The Analysis of the Attack" Let's try to analyze this attack and try to find some possible ways to prevent it. First, this attack exploits the fundamental weaknesses of some Windows components, which have not been changed for years. It means, that all Windows OSes since NT 4.0 are vulnerable to this attack, even the newest Windows 10.	
Slide 34 "The Analysis of the Attack" Moreover, I managed to buy a book, which was the first that described the Security Reference Monitor and the Object Manager. So, the attack we are talking about is applicable since 1993 and this is one of the oldest vulnerabilities **/ˌvʌln(ə)rəˈbɪlətis/** ever found. When I found this vulnerability, I felt like an archeologist, who had revealed a new ancient artefact. But, {big-pause} the challenge of memory protection is much much older.	

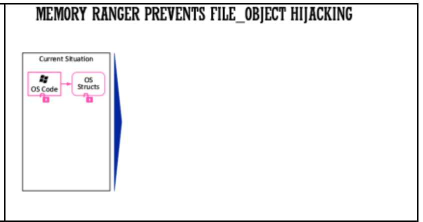
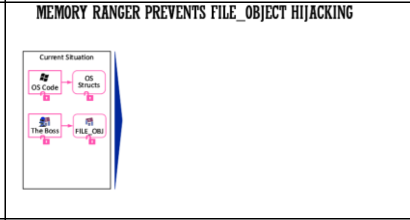
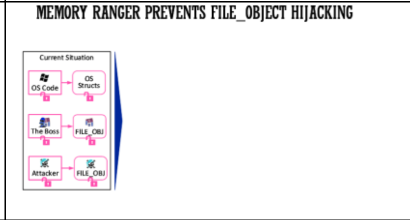
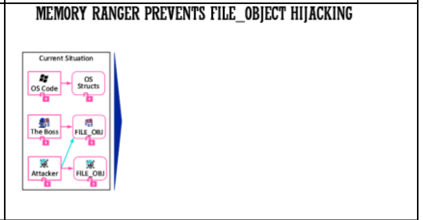
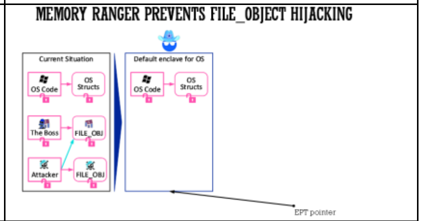
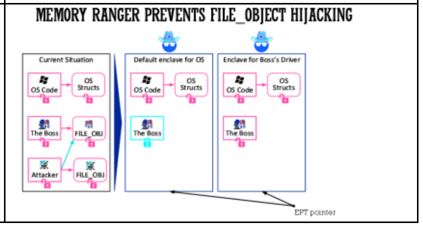
Slide 35 “The Analysis of the Attack” The very first mention of the illegal memory access and ways of preventing them dates back to 1965. Multics system was developed to isolate memory. You can see that authors of Multics were focused on memory protection for General Electric mainframes. That was the first generation of mainframes and they already had security problems. Could you imagine that a half-century has passed and little has changed? And we still have this problem with memory access restriction.	
Slide 36 “The Analysis of the Attack” So, let’s go back to nowadays and think how can we protect FILE_OBJECT structures?	
Slide 37 “The Analysis of the Attack” To provide confidentiality we usually apply encryption.	
Slide 38 “The Analysis of the Attack” But {-pause-} the problem is that these structures are allocated dynamically. To protect FILE_OBJECT we need to encrypt their content.	
Slide 39 “The Analysis of the Attack” Next, we need to trap any access to this memory	

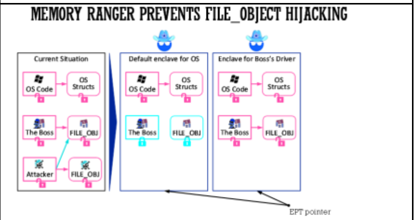
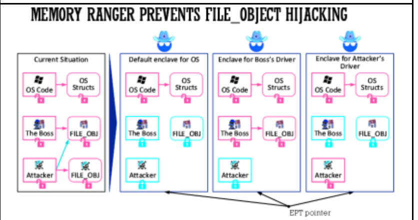
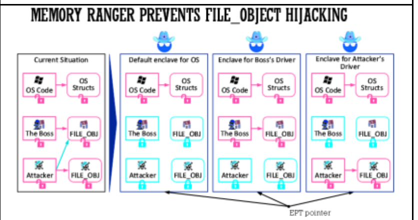
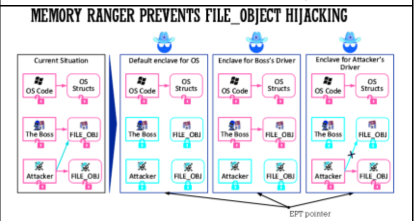
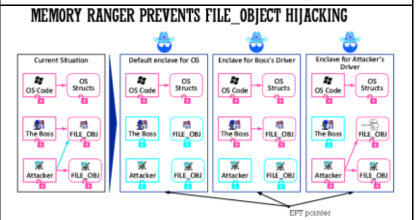
Slide 40 “The Analysis of the Attack” in order to decrypt its content, allow **/əˈlaʊ/** access,	
Slide 41 “The Analysis of the Attack” wait until the file operation is completed,	
Slide 42 “The Analysis of the Attack” and encrypt FILE_OBJECT again. We need to repeat these steps	
Slide 43 “The Analysis of the Attack” for each read and write operation and for each file. And it is absolutely obvious that, such encryption will cause huge	
Slide 44 “The Analysis of the Attack” performance degradation.	
Slide 45 “The Idea of MemoryRanger” I think the main problem of Windows kernel security is that all drivers and OS kernel	

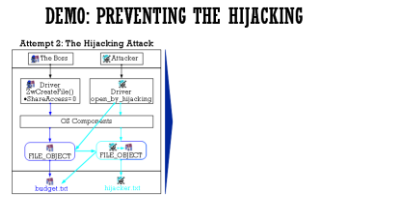
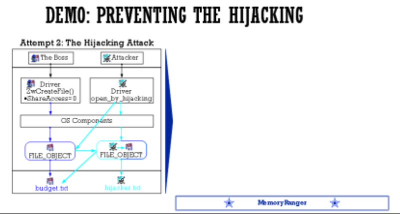
Slide 46 “The Idea of MemoryRanger” share the same memory space.	
Slide 47 “The Idea of MemoryRanger” If we could move these three drivers into three isolated memory enclosures, we would protect their memory from each other.	
Slide 48 “The Idea of MemoryRanger” But, the thing is that Windows OS doesn’t give us such an opportunity.	
Slide 49 “EPT main mechanism” Luckily for us, hardware virtualization technology <u>provídes</u> the mechanism, which can be used to implement the idea of memory ísolátion /ˌaɪsəˈleɪʃ(ə)n/. I mean Extended Page Tables or EPT feature.	
Slide 50 “EPT main mechanism” Without EPT guest physical addrésses are used to access the host physical memory. Address H is equal to the address G.	
Slide 51 “EPT main mechanism” Using EPT we can get an additional or second level of address translation. When EPT is enabled, guest physical addresses are translated to host physical addrésses by traversing a set of EPT paging structures.	

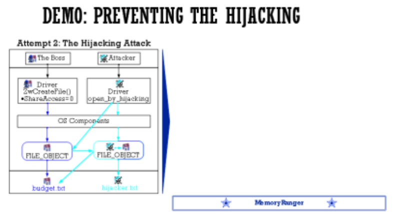
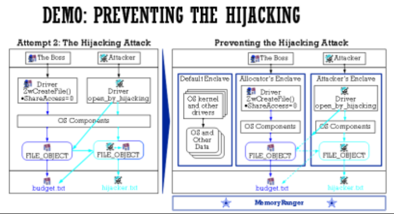
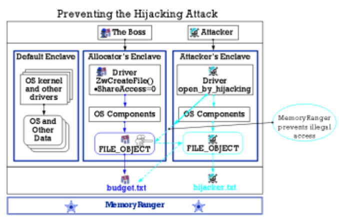
Slide 52 “EPT main mechanism” Now, address H is calculated using EPT mechanism. Let’s have a look at EèePèeTèe details.	
Slide 53 “EPT main mechanism” The key component of EPT mechanism is the set of EPT paging structures. These structures include EPT entries, which determine	
Slide 54 “EPT main mechanism” the mapping between the guest memory and the host memory. Let’s see how we can apply EPT.	
Slide 55 “EPT three main features” First, EPT can help to control read, write and even execute access for each memory page.	
Slide 56 “EPT three main features” For example, to trap any read access to the sensitive data	
Slide 57 “EPT three main features” we set to the zero the corresponding access bits on page with this data. Also, after trapping such access attempts,	

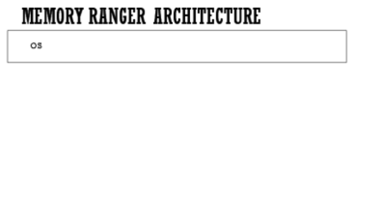
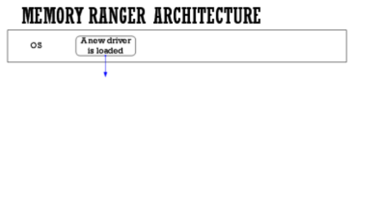
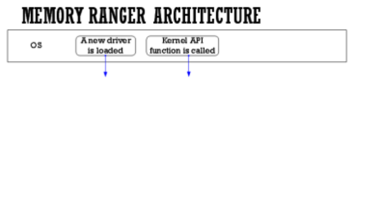
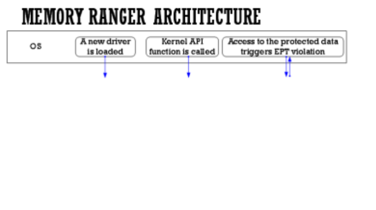
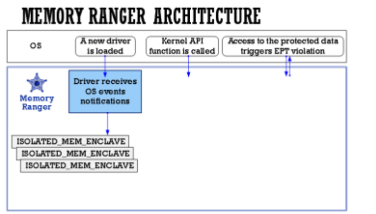
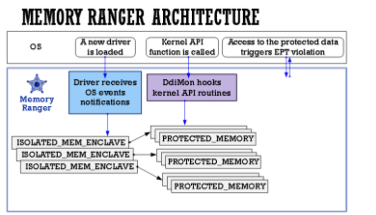
<table>
<tr><td>Slide 58 “EPT three main features”
we can redirect /ˌriːdɪˈrekt/ them from the secret page to the new memory page a page with fake data.</td><td>
</td></tr>
<tr><td>Slide 59 “EPT three main features”
a page with fake data</td><td>
</td></tr>
<tr><td>Slide 60 “EPT three main features”
Second, these /ðiːz/ <u>memory access permissions</u> and <u>redirected confígurátions</u> /kənˌfɪgjəˈreɪʃ(ə)nz/ can be changed in real time. So, we can dynamically restrict access to any memory regions using one set of EPT paging structures.</td><td>
</td></tr>
<tr><td>Slide 61 “EPT three main features”
Finally, we can állocáte /ˈæləkeɪt/ several sets of EPT paging structures with various memory access confígurátions /kənˌfɪgjəˈreɪʃ(ə)nz/. And by switching between them, we can organize drivers’ execution in ìsolat<u>ed</u> enclaves. That’s exactly what we need.</td><td>
</td></tr>
<tr><td>Slide 62 “EPT three main features”
The main idea of MemoryRanger</td><td>
</td></tr>
<tr><td>Slide 63 “EPT three main features”
is to /ˈsepəreɪt/ séparáte drivers in that way, so all drivers will be executed in isolated memory enclosures. Let me show you how to implement this idea to protect FILE_OBJECT structures.</td><td>
</td></tr>
</table>

Slide 64 "Basic scenario of MemoryRanger" I explored the following scenario. The OS kernel is loaded into the memory and accesses its data.	
Slide 65 "Basic scenario of MemoryRanger" After that the boss loads its driver, and opens the budget file. You can see the corresponding FILE_OBJECT structure.	
Slide 66 "Basic scenario of MemoryRanger" Next the attacker's driver is loaded and it creates the file hijacker.	
Slide 67 "Basic scenario of MemoryRanger" Also, the attacker tries to access the budget FILE_OBJECT structure in order to reveal it. As you remember that all drivers share the same memory space and such an illegal <u>access attempt</u> is becoming uncontrolled. Let's roll back and launch my MemoryRanger to isolate all these drivers from the rest of the OS kernel.	
Slide 68 "Basic scenario of MemoryRanger" First of all, MemoryRanger allocates the EPT structure for the default enclave. Windows OS kernel is executed inside the default enclave.	
Slide 69 "Basic scenario of MemoryRanger" Next, MemoryRanger traps the loading of the boss's **/bɒses/** driver and allocates a new EPT structure for the boss's enclave. MemoryRanger updates all EPT structures in the following way. • Only boss's driver and OS kernel are executed inside boss's enclave.	

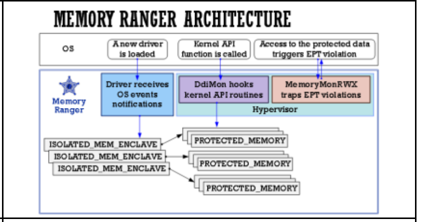
• In the default enclave, MemoryRanger restricts access to the boss’s driver. This memory access confígurátion /kənˌfɪɡjəˈreɪʃ(ə)n/ guarantees that the boss’s driver is executed inside its enclave only.	
Slide 70 “Basic scenario of MemoryRanger” The boss’s driver creates a budget file, the OS allocates the corresponding FILE_OBJECT structure. My MemoryRanger updates all enclaves’ **/kənˌfɪɡjəˈreɪʃ(ə)nz/** confígurátions again. As a result, the boss’s driver áccessés the budget only inside its enclave. MemoryRanger traps **/træps/** the loading of the attacker’s driver	
Slide 71 “Protecting FILE_OBJECTs by MemoryRanger” and creates an enclave for it. You can see that the attacker’s driver can be executed only inside its own enclave, where the budget FILE_OBJECT structure is protected. The boss’s driver is protected as well.	
Slide 72 “Protecting FILE_OBJECTs by MemoryRanger” Then the attacker’s driver creates the file hijacker. MemoryRanger traps this event as well and updates all enclaves’ /kənˌfɪɡjəˈreɪʃ(ə)nz/ confígurátions again.	
Slide 73 “Protecting FILE_OBJECTs by MemoryRanger” Finally, the attacker’s driver tries to read the FILE_OBJECT structure of the budget file. This event causes EPT memory violation, which is processed by my MemoryRanger.	
Slide 74 “Protecting FILE_OBJECTs by MemoryRanger” MemoryRanger is foisting the fake page on the attacker instead of the real one. In the same way, MemoryRanger allocates a separate EPT structure or memory enclave for each newly loaded driver and updates all other enclaves. This is the main idea. Let’s move on to the demonstration.	

Slide 75 “Demo: The Attack Prevention” The boss will load its driver to draw up the budget. After that, the attacker’s driver will be loaded in order to hijack the budget.	
Slide 76 “Demo: The Attack Prevention” Let’s have a look what will happen if I launch MemoryRanger befo-o-re.	
Slide 77 “Demo: The Attack Prevention” First of all, we are launching MemoryRanger console application. It loads the hypervisor /ˈ**haɪpə(r)ˌ vaɪzə(r)/** to protect kernel memory. Then, the boss is launching his console, which loads a driver. The boss creates the budget file. And we can see the created file handle and the allocated FILE_OBJECT structure. By using **f_write** the boss is setting up the budget. Now, the boss is checking the budget. Done. Now the boss receives a call and the budget file remains open. At this moment, the attacker is getting a chance to snoop the budget. He is launching its console. He is trying to open the budget using file system routine ZwCreateFile. So, he fails to open the file, because Windows prevents illegal access to the budget file. The attacker is willing to gain an access by hijacking. He copies the address of the FILE_OBJECT. By using **f_open_by_hijacking** the attacker creates a file hijacker. Windows creates this file and returns the file handle and the FILE_OBJECT structure. The attacker is trying to read the budget FILE_OBJECT, but he fails. MemoryRanger prevents illegal memory access to the budget FILE_OBJECT. Anyway, the attacker is still hoping to read the budget.	

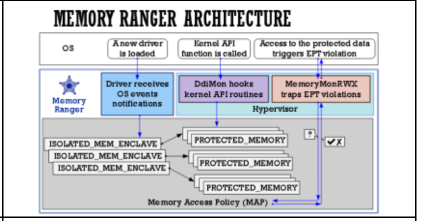
But instead of reading a real budget, the attacker can read only the deliberately foisted fake {pause} null data. He fails again. But anyway, he is trying to overwrite the budget file. If the boss decides to check his budget he will see the originals data. MemoryRanger prevents all illegal access attempts to the budget. Finally, let's compare these **/ðiːz/** two files. We close the budget file and the file hijacker and both control consoles as well. We can see that the budget file includes only the budget and all attackers input data is in his file. The OS is protected.	
Slide 78 "Demo: The Attack Prevention" So, at the begging we see that situation. Without MemoryRanger, the attacker can access FILE_OBJECT structure without any issues.	
Slide 79 "Demo: The Attack Prevention" Here you can see the difference between Windows without MemoryRanger and wíth it. MemoryRanger prevents illegal memory access by running drivers into ísoláted kernel enclosures **/ɪnˈkləʊʒə(r)z/**. MemoryRanger traps and controls memory access attempts to the sensitive data.	
Slide 80 "Demo: The Attack Prevention" Here is the detailed scheme **/skiːm/** , after two drivers have been loaded. MemoryRanger step-by-step allocates three memory enclosures for the OS kernel, for the boss's driver, and finally for the attacker's driver. That scheme helps to intercept and prevent illegal memory access attempts. Actually, it looks very similar with the scheme, which was proposed in the paper from 1965. Now let's move on to the MemoryRanger architecture.	

Slide 81 “Requirements for MemoryRanger” We need to process the following three events:	
Slide 82 “Requirements for MemoryRanger” loading of a new driver	
Slide 83 “Requirements for MemoryRanger” calling kernel API functions and	
Slide 84 “Requirements for MemoryRanger” processing memory access violations, which occur due to the access to the memory with restricted access.	
Slide 85 “Demo: The Attack Prevention” After a new driver has been loaded MemoryRanger needs to create an enclave for it and updates memory access configurations. The information about the newly created enclave is saved into ISOLATE_MEM_ENCLAVE structure. To be notified whenever a new driver is loaded MemoryRanger uses the callback routine.	
Slide 86 “Demo: The Attack Prevention” After trapping kernel functions such as ZwCreateFile MemoryRanger locates the created FILE_OBJECT structure and modifies the corresponding memory access permission. The information about the protected memory regions is saved into PROTECTED_MEMORY structures. To hook these routines, MemoryRanger leverages DdiMon component. This component can trap any kernel API function transparently for the OS.	

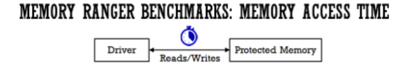
Slide 87 “Demo: The Attack Prevention”
The third component is MemoryMonRWX, which is able to track and trap all types of memory access: read, write, and execute. MemoryMonRWX processes EPT violations. Execute violation means that it is time to change enclaves so another driver will continue its execution inside its enclave. Read or write violation means that this access is illegal and MemoryMonRWX redirects this violation to the Memory Access Policy.



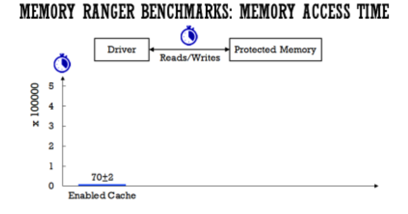
Slide 88 “Demo: The Attack Prevention”
Memory Access Policy (MAP) plays a role of an intermediary during memory access to the protected data and decides whether to block or allow this access. Memory Access Policy relies on the ISOLATE_MEM_ENCLAVE structures and PROTECTED_MEMORY structures.
Now let’s move on to the benchmark assessment.



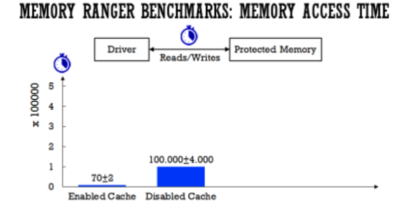
Slide 89 “Benchmarks”
☺ I measured the time of legal access from the driver to the protected data in four situations.



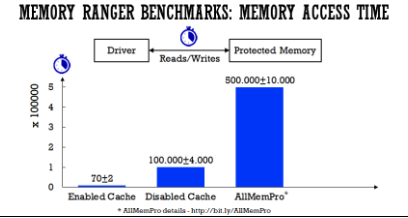
Slide 90 “Benchmarks”
memory protection with enabled cache,



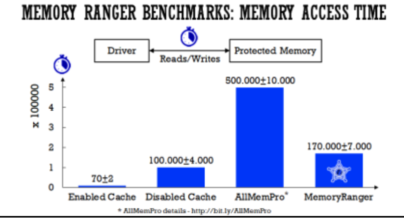
Slide 91 “Benchmarks”
with disabled cache, and with two memory protectors:



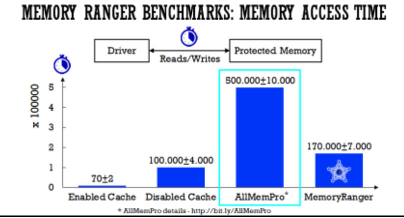
Slide 92 “Benchmarks”
AllMemPro



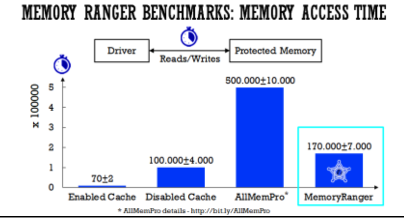
Slide 93 “Benchmarks”
and MemoryRanger.



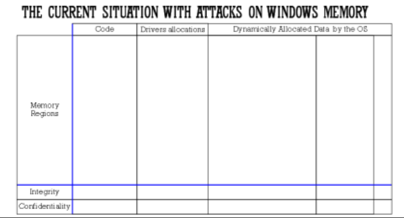
Slide 94 “Benchmarks”
AllMemPro or Allocated Memory Protector is the closest competitor for MemoryRanger, which I developed a year ago and presented at the previous ADFSL conference in Texas. AllMemPro uses only one EPT structure to prevent illegal access to the allocated memory.



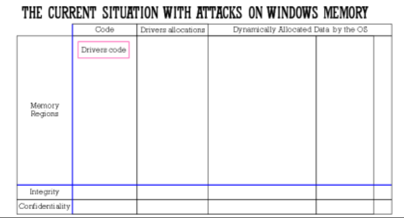
Slide 95 “Benchmarks”
We can conclude that MemoryRanger is a bit slower than access to the memory with disabled cache and it three times faster than the closest competitor.



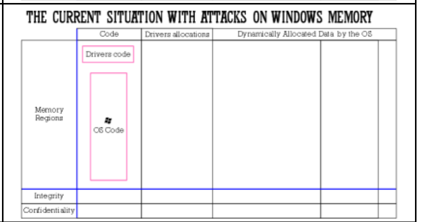
Slide 96 “Windows Kernel memory”
So let’s briefly go for the sensitive memory areas, which can be used during cyber-attacks.



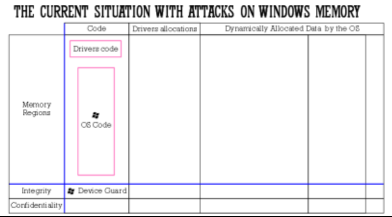
Slide 97 “Windows Kernel memory”
Here we have the third-party drivers



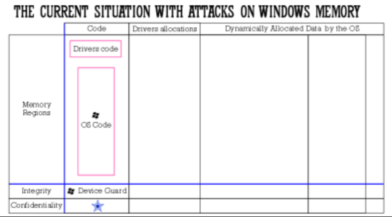
Slide 98 “Windows Kernel memory”
and OS kernel code.



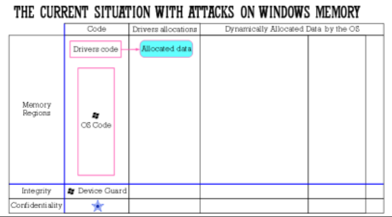
Slide 99 “Windows Kernel memory”
Device Guard provides the integrity for all code sections. But Windows does not provide the code confidentiality.



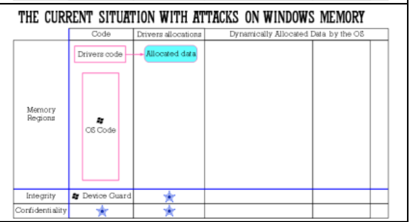
Slide 100 “Windows Kernel memory”
MemoryRanger fills this gap.



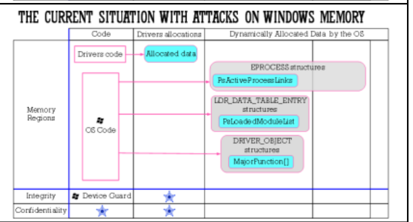
Slide 101 “Windows Kernel memory”
Next, we have allocated data by the third-party drivers. Windows does not protect that memory.



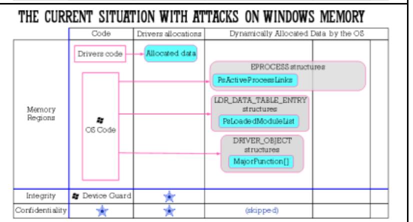
Slide 102 “Windows Kernel memory”
My MemoryRanger can fill both these gaps as well.



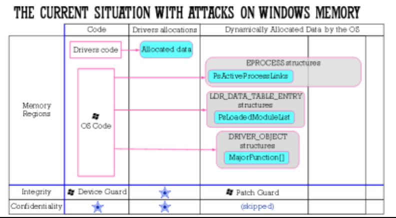
Slide 103 “Windows Kernel memory”
Now, let’s move on to the <u>internal</u> allocated memory.



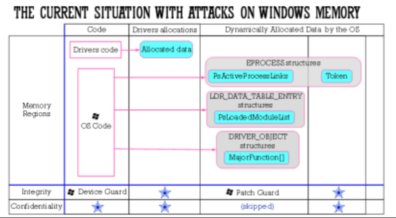
Slide 104 “Windows Kernel memory”
I do not think, that their confidentiality is really needed, so let’s skip it.



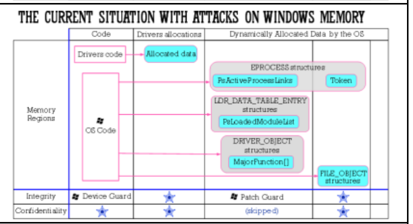
Slide 105 “Windows Kernel memory”
PatchGuard provides the integrity of these internal data regions, but only partially.
For example, PatchGuard does not protect EPROCESS structures completely.
It prevents hiding process by checking the PsActiveProcessLinks field.



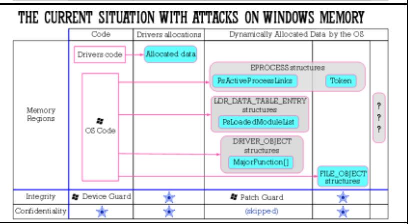
Slide 106 “Windows Kernel memory”
But recent privilege escalation attacks show that the Token field of EPROCESS structure is not protected.
MemoryRanger can protect Token field as well.



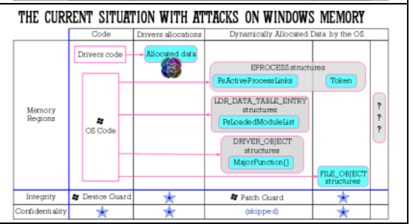
Slide 107 “Windows Kernel memory”
Today I've presented that FILE_OBJECT structure is also vulnerable. MemoryRanger fill this gap too. It protects the integrity and confidentiality of FILE_OBJECTs.



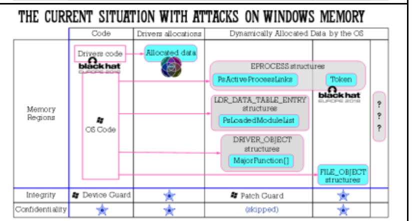
Slide 108 “Windows Kernel memory”
Currently, I’m focusing on some new memory regions, which must be protected.

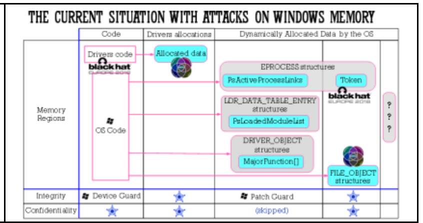
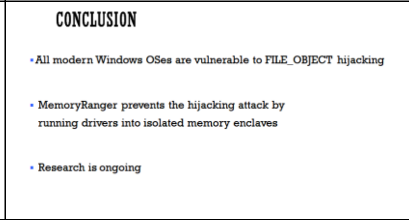


Slide 109 “Windows Kernel memory”
A year ago, at the ADFSL conference, I presented how dynamically allocated data can be protected.



Slide 110 “Windows Kernel memory”
At the BlackHat I presented how to prevent stealing drivers code and protect Token field into EPROCESS structure.



Slide 111 “Windows Kernel memory” Today I’ve demonstrated you how to prevent illegal access to the FILE_OBJECT structures.	
Slide 112 “Conclusion” Let me recap very briefly on what we’ve done. First of all, we have seen that files open in an exclusive mode can be illegally accessed without any security reaction. After that, I’ve presented my MemoryRanger, which can prevent such an unauthorized memory access. The further research is ongoing.	
Slide 113 “Thank you” Thank you!