

Divide et Impera: MemoryRanger runs drivers in Isolated Kernel Spaces

Slide 1 Hello

Hi! I'm Igor!

Today I'd like to show, to present you my MemoryRanger, which can isolate drivers code and allocated data from illegal access in the kernel memory. This topic is of critical importance for all security experts who are dealing with protection of operating systems.



Slide 2 “WhoamI”

A few words about mé. I earned my PhD 7 years ago. My area of expertise includes Windows Kernel security, rootkits detection and hypervisors /'haɪpə(r),vaɪzə(r)z /.

I carry out research projects by my own as a hobby. I'm a fan of cross-disciplinary research. You can find my results in my blog. By the way I am fond of travelling and powerlifting as well.

WHOAMI

- MEFH Alumni, PhD in Cyber Security, published 23 papers
- Area of interest is Windows Kernel security:
 - Memory Forensics
 - Rootkits Detection
 - Bare-Metal Hypervisors
- Fan of academic cross-disciplinary research - igorkorkin.blogspot.com
- Love traveling and powerlifting - @igor.korkin

Slide 3 “Agenda and general questions”

I have divided my talk into three main parts. First we'll be looking at the consequences of attacks on kernel memory. After that, I'll give you a brief analysis of Windows built-in security features and research projects, which are designed to protect kernel memory. Finally, I'll show you how I solve this problem using my MemoryRanger. Feel free to ask me any questions at the end of the talk.

AGENDA

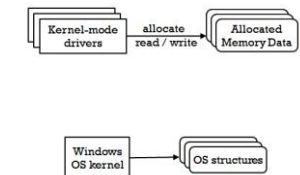
- Attacking the kernel-mode memory
- Existing protection: Windows built-in security and research projects
- MemoryRanger hypervisor: idea, details, demos

Slide 4 “Problem and consequences”

Nowadays kernel-mode drivers share the same memory space with the rest of the OS kernel.

All drivers can read and write any part of kernel-mode memory without any hardware restrictions; this fact makes the modern operating systems to be prone to rootkit attacks and kernel exploitation.

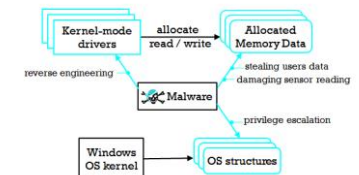
ATTACKS ON KERNEL MODE MEMORY

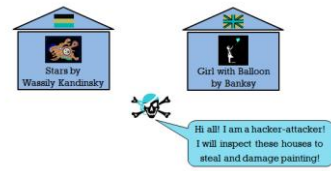
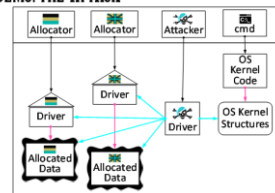


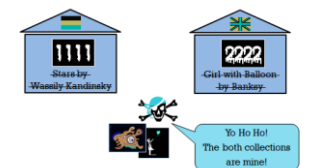
Slide 5 “Problem and consequences”

Malware driver can dump the drivers' code and tamper allocated memory, for example, Windows Internal structures and allocated data of the third party drivers. As a result, hackers can escalate privileges, hide their footprints, steal users' secrets, or even disrupt industrial systems, which are controlled by drivers. Let me demonstrate you some of these attacks.

ATTACKS ON KERNEL MODE MEMORY



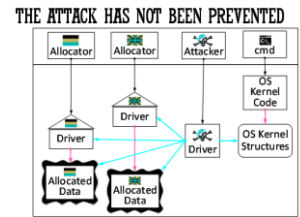
<p>Slide 6 “Demo: The Attack”</p> <p>Let’s imagine two houses with private art collections: Russian house and a British one.</p> <p>In the Russian house we can see the panting of Wassily Kandinsky, you know he was a great Russian painter. The Banksy’s art is in the British house. I hope everyone is familiar with Banksy’s painting.</p> <p>And here is the “<u>hacker-attacker</u>”, who wants to grab all these masterpieces. If <u>these</u> houses are not protected the both private collections will be stolen. In the kernel-mode memory, the situation is absolutely the same.</p>	<p>TWO HOUSES WITH PRIVATE ART COLLECTIONS</p> 
<p>Slide 7 “The attack scenarios”</p> <p>If drivers are not protected the data can be stolen. Let’s see how it can happen.</p> <p>I will load two drivers. The Russian driver and a British one, which store their data in the allocated memory pools. After that, the attacker’s driver will be loaded to dump the drivers’ code and steal both Russian and British data buffers. And finally, I will show you how the attacker can even escalate process privileges.</p> <p>Let’s see how it can happen.</p>	<p>DEMO: THE ATTACK</p> 
<p>Slide 8 “ (Demo: The Attack)</p> <p>I am launching a Russian console /kən'səʊl/, which loads a Russian driver. In the same way, I am loading a British driver.</p> <p>Finally, the hacker-attacker is launching its driver to access kernel memory.</p> <p>Now, the attacker is trying to dump the Russian driver using its loaded address. Look, the attacker is successfully reading the first byte, which is M and the second one is Z.</p> <p>What about the British driver? Hackers are reading the first two bytes without any problems as well.</p> <p>We can see that the <u>code</u> of both drivers can be <u>stolen</u>.</p> <p>Let’s move on to the private <u>data</u>, which can be stored in the allocated memory.</p> <p>The Russian driver is allocating data and setting it content as “Wassily Kandinsky”.</p> <p>Let’s check “Kandinsky”. Fine. The data has been allocated.</p> <p>The British driver is allocating memory and setting Banksy as its content.</p> <p>Let’s check Banksy. It’s all right.</p> <p>Now the attacker wants to steal these “<u>priceless</u>” data content.</p>	<p>DEMO: THE ATTACK</p> <p>The online version is here – https://www.youtube.com/embed/HNac-ty3QA?rq=hd1080</p>

<p>To steal Kandinsky, hackers need to know the corresponded memory address. Let's assume they do know this address. Usually hackers do know everything. So the attacker can read this data, and overwrite it. Here we have <u>only four ones</u> instead. We can see that the attacker did modify the data. Now, the attacker is entering the British house to <u>steal</u> Banksy's art. After overwriting we have only <u>four twos</u> instead. Banksy collection has disappeared. Both data buffers were <u>stolen</u> and <u>modified</u>.</p>	
<p>Slide 9 "The Result With Houses" Hacker-attacker has successfully stolen both private collections. It means that our houses were not protected in a proper way. What was wrong? How can we defend them? We will see it later. And now let's go back to demo to see how the attacker can even escalate process privileges.</p>	<p>TWO HOUSES WITH PRIVATE ART COLLECTIONS</p> 
<p>The attacker is starting the CMD. After launching it, he is getting its process ID. Done. He is copying its PID. We can see that the CMD has users' privileges and it is not enough to reconfigure the Windows Firewall, which is expected. Now the attacker is trying to elevate the CMD privileges. He is using PRIV command with process ID. Checking the privileges again. The privileges have been escalated. Now the CMD has the highest privileges and now CMD can reconfigure the Windows Firewall service and even can turn it off. We can see only a small message, which confirms that the Firewall is out of service and it cannot be automatically restarted. This is very good news for hackers. But let's wait for PatchGuard reaction, which is designed to prevent illegal memory modifications. We've been waiting for 10 hours it is quite a long time and nothing has happened. The OS has not been crushed. It means that PatchGuard didn't prevent the invasion and the OS became infected.</p>	

Slide 10 “The Result With Drivers”

So, we can see that Windows Kernel cannot prevent illegal memory access. The attacker can read the driver’s code, steal and change allocated data and even escalate process privileges without any security reaction, such as a Blue Screen of Death.

Let’s have a look at the current situation with memory protection projects.



Slide 11 “Current situation with memory protection”

Look at this table. Microsoft has developed a number of security features for memory protection. The first one is “Device Guard”, which provides kernel code integrity by marking executable pages as “read-only”. So any modifications of the code will crash the system. Not bad.

The second one is “PatchGuard”, which crashes the OS after revealing some changes of internal structures, for example EPROCESS unlinking. But you’ve just seen that PatchGuard does not protect EPROCESS structures completely. Finally, Windows Security does not protect memory allocated by the third party drivers.

Many security experts from all around the world are trying to fill /fil/ this gap. There are several research projects, which are designed for memory protection. And you can see that there is no general solution, which can deal with all these issues at the same time. Today I will propose the solution. I’ll present you my MemoryRanger, which provides both integrity and con-fiden-tiality for drivers code and allocated data. ☺

BACKGROUND ANALYSIS

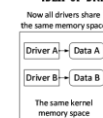
Memory protection projects	Malware attacks on					
	Code of OS & third-party drivers		OS data: internal structures		Data of third-party drivers	
	Read	Write	Read	Write	Read	Write
Windows Security	-	BSOD 0x8E by Device Guard	-	BSOD 0x109 by PatchGuard	-	-
PrivGuard	-	-	-	+	-	-
LAKEED	+	+	+	+	-	-
IKMG	-	+	+	+	+	+
rR*X	+	+	-	-	-	-
AllMemPro	-	-	+	+	+	+
Memory Ranger	+	+	+	+	+	+

Slide 12 “Thinking about separate drivers execution”

Look at this.

Nowadays kernel mode drivers share the same memory space with the rest of the kernel.

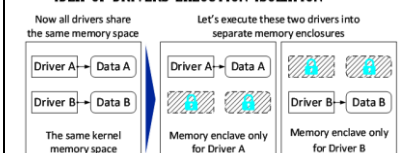
IDEA OF DRIVERS EXECUTION ISOLATION

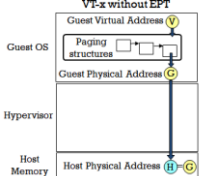
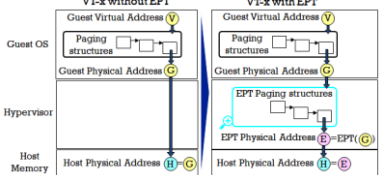
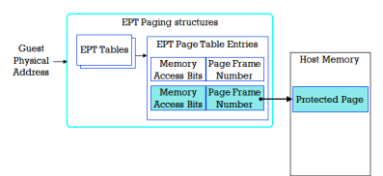
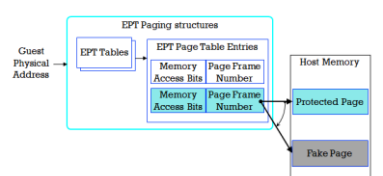
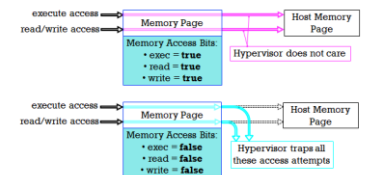


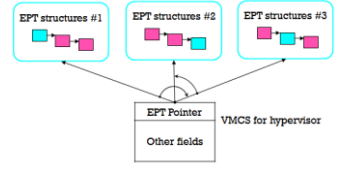
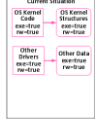
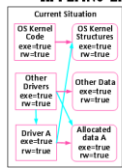
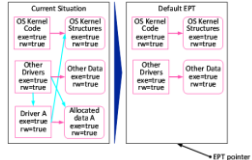
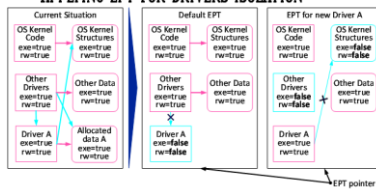
Slide 13 “Thinking about separate drivers execution”

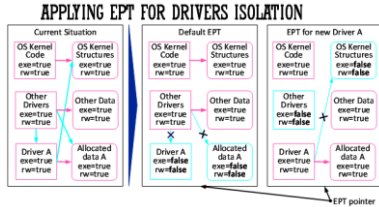

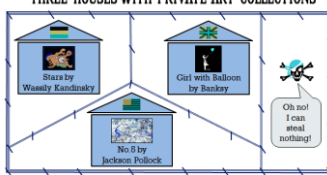
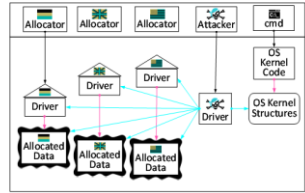
If we can move these two drivers with their allocations into two separate memory enclosures, we will protect their memory from each other. The thing is that Windows OS doesn’t give us such an opportunity.

IDEA OF DRIVERS EXECUTION ISOLATION



<p>Slide 14 “EPT main mechanism”</p> <p>Luckily for us, hardware virtualization technology provides the mechanism, which can be used to implement the idea of memory isolation /aɪsə'leɪʃ(ə)n/.</p> <p>I mean Extended Page Tables or EPT feature. Without EPT guest physical addresses are used to access the host physical memory. Address H is equal to the address G.</p>	<p>PROCESSING MEMORY ACCESS: EPT FEATURE</p> <p>VT-x without EPT</p> 
<p>Slide 15 “EPT main mechanism”</p> <p>Using EPT we get an additional or second level of address translation. When EPT is enabled, guest physical addresses are translated to host physical addresses by traversing a set of EPT paging structures. Now, address H is equal to the address E, not to G. Let’s have a look at EPT details.</p>	<p>PROCESSING MEMORY ACCESS: EPT FEATURE</p> <p>VT-x with EPT</p> 
<p>Slide 16 “Details about EPT bits”</p> <p>EPT structures include EPT entries, which determine the mapping between the guest memory and the host memory. Each EPT entry includes memory access bits and page frame number.</p>	<p>INSIDE EPT PAGING STRUCTURES. EPT PFN</p> 
<p>Slide 17 “Details about EPT bits”</p> <p>This number is linked to the corresponding host physical address. It can be used to prevent access to the protected data. We can redirect /rɪ'daɪrɛkt/, the access from the protected data to the <u>fake one</u> by changing PFN value to the address of fake page.</p>	<p>INSIDE EPT PAGING STRUCTURES. EPT PFN</p> 
<p>Slide 18 “Details about EPT bits”</p> <p>Access bits indicate whether access attempts are allowed to the page referenced by this entry.</p> <p>Any <u>disallowed</u> access attempts will trigger EPT violations, and then control goes to the hypervisor.</p> <p>So we can trap read- write and execute- access to the page by changing these memory access bits.</p>	<p>INSIDE EPT PAGING STRUCTURES. EPT BITS</p> 

<p>Slide 19 “Details about EPT bits”</p> <p>Another interesting feature is that we can allocate several EPT paging structures with various memory access configurations. By switching between them, we can organize drivers’ execution in <u>isolated enclaves</u>. EPT feature seems to be very promising for the memory isolation /,aɪsəˈleɪʃ(ə)n/. Let me show you how all these ideas are implemented in my Memory Ranger.</p>	<p>INSIDE EPT PAGING STRUCTURES</p> 
<p>Slide 20 “Idea of Applying EPT”</p> <p>I explored the following scenario. The OS kernel and other drivers are loaded into the memory and access their data. We have two legal access attempts, which are marked by green arrows.</p>	<p>APPLYING EPT FOR DRIVERS ISOLATION</p> 
<p>Slide 21 “Idea of Applying EPT”</p> <p>As the Driver A is loaded, it allocates memory and accesses this newly allocated data. All in all we have three legal access attempts, which are marked by green. Red lines indicate the following illegal access attempts. Driver A modifies the OS internal structures. Other Drivers read the memory of Driver A and try to overwrite its allocated data. As you remember that all drivers share at the same memory space, all these illegal access attempts are becoming uncontrolled.</p>	<p>APPLYING EPT FOR DRIVERS ISOLATION</p> 
<p>Slide 22 “Idea of Applying EPT”</p> <p>Let’s roll back and launch MemoryRanger to isolate Driver A from the rest of the OS kernel. First of all, my MemoryRanger allocates the default EPT structure. All loaded drivers and Windows Kernel are executed inside the default EPT structure.</p>	<p>APPLYING EPT FOR DRIVERS ISOLATION</p> 
<p>Slide 23 “Idea of Applying EPT”</p> <p>MemoryRanger traps the loading of Driver A and allocates a new EPT structure for Driver A. Memory Ranger updates all EPT structures in the following way.</p> <ul style="list-style-type: none"> • Only Driver A and OS kernel are executed inside new EPT. • In the new EPT structure, memory access bits have been cleared to 0 for OS structures to restrict access from Driver A. • In the default EPT structure, memory access bits for the driver A have been cleared as well. 	<p>APPLYING EPT FOR DRIVERS ISOLATION</p> 

<p>Slide 24 “Idea of Applying EPT”</p> <p>After driver A allocates memory data, my Memory Ranger updates all EPT structures again. As a result, <u>any access</u> from other <u>drivers</u> to the <u>driver A</u> and its <u>allocated memory</u> will be <u>trapped</u> and <u>blocked</u>. Also, all access to the OS structures is forbidden for the Driver A.</p> <p>In the same way, Memory Ranger allocates a separate EPT structure for each new driver and updates all other EPT structures. This is the main idea. Let’s move on to the demonstration.</p>	
<p>Slide 25 “Demo the Attack Prevention”</p> <p>Let’s add one more treasure house to our criminal story, let it be American house full of Jackson Pollock paintings. Now we have three houses with three private art collections. The attacker is more motivated. ☺</p> <p>To protect these houses we need to build at least a fence.</p>	
<p>Slide 26 “Demo the Attack Prevention”</p> <p>It helps to prevent attacker’s access and isolates three houses from each other as well.</p>	
<p>Slide 27 “Demo the Attack Prevention”</p> <p>In the kernel mode, the drivers memory will be isolated in the same way. I will load three drivers. We call them Russian, British and American.</p> <p>All these drivers store their data in the allocated memory pools. After that, I will launch a <u>malware driver</u> in order to steal and modify <u>these</u> data buffers as well as dumping the drivers’ code. Finally, a malware driver will try to escalate process privileges.</p> <p>Let’s have a look what will happen if I launch MemoryRanger befo-o-re.</p>	
<p>Slide 28 “Demo (The Attack Prevention)”</p> <p>First of all, I am launching Memory Ranger console application. It loads the driver and activates the hypervisor.</p> <p>Then, I am launching three drivers: Russian, British, and American.</p> <p>Finally, the hacker-attacker is launching its driver.</p>	<p>DEMO: THE ATTACK PREVENTION</p> <p>The online version is here – https://www.youtube.com/embed/vrm8cgn5Dsl?yq=hd1080</p>

The attacker wants to dump the Russian driver first. He copies its address and we can see, that the attacker can read zero value. What about the next byte? He has a zero value as well.

So the attacker fails to dump this driver.

Now let's read the drivers' content using the driver itself. Now we have M and Z value.

We can see that Memory Ranger can protect the content of the Russian driver.

Now the attacker is entering the British house. He is going to dump British driver.

He is copying its address and trying to read its content. So he gets a zero value again.

MemoryRanger is foisting the fake null data to the attacker instead of the real one.

Let's try to access the British driver using the British driver itself. Now we are reading MZ bytes.

It means that MemoryRanger can prevent unauthorized access attempts and grants the legal access.

☺ And now let's check if the American driver can read the content of the British driver. We have zero value again. <MSC>Memory Ranger isolates drivers from each other</MSC>.

Let's move on to the private data, which can be stored in the allocated memory.

For the Russian driver I am allocating "Wassily Kandinsky".

Now, let's check the allocation. Wassily Kandinsky.

For the British driver I am allocating "Banksy".

Now, let's check the allocation. Banksy.

And finally for the American driver I am allocating the data and setting "Jackson Pollock".

Now, let's check this allocation. Jackson Pollock.

Now the attacker wants to steal all these treasures. Let's see if he does manage that.

He is starting with Russian house and Kandinsky art. He copies its address and tries to read.

The attacker can read only a null string. He is trying to overwrite this data.

It's time to check if the Kandinsky has been modified. We go back to the Russian driver and check the allocation. Memory Ranger is protecting this data. Hackers failed to steal and modify Kandinsky.

Now the attacker is entering the British house to steal Banksy. He is coping its address. And we can see that the attacker have read nothing again. He is trying to overwrite this data.

Now, we go back to the British app and check Banksy collection.

We can see that the MemoryRanger is protecting Banksy as well.

And now let's check if the American driver can access Banksy's data. As you can see the American driver can read nothing. It is trying to overwrite the data. No results. Memory Ranger is isolating drivers allocation from each other as well.

We can change the data only by the driver, which allocates this véry dáta. Let's check it.

Let's replace "Wassily Kandinsky" with "Ivan Aivazovsky".

The allocated buffer has been updated in a legal way.

MemoryRanger is protecting even allocated data.

Let's start CMD to test privilege escalation attack.

The attacker is launching CMD, getting its process ID, and checking its privileges. As we expected CMD has users' privileges and cannot reconfigure the Windows Firewall.

The attacker is trying to elevate the CMD privileges. The attacker is using PRIV command with process ID to escalate privileges.

Let's check the privileges. Cmd has still users' privileges and cannot reconfigure the Windows Firewall service. It means that attacker has failed. CMD is not able to stop the Windows Firewall service.

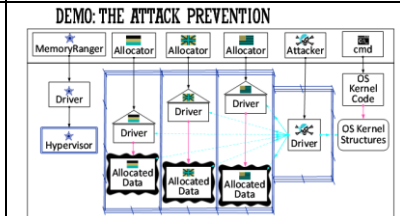
The OS is protected.

Slide 29 "Demo the Attack Prevention"

So you have just seen that MemoryRanger prevents illegal memory access by running drivers into isolated kernel spaces or enclosures */ɪnˈkləʊʒə(r)z/*. Let's have a look at memory access rules which are provided by MemoryRanger.

Slide 30 "Demo the Attack Prevention Final"

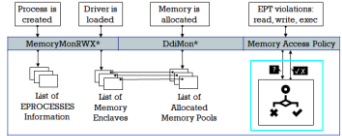
Memory Ranger protects memory according to the principle of least privilege. Newly loaded drivers can access only their own code.

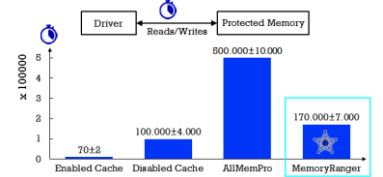


MEMORY RANGER: PRINCIPLE OF LEAST PRIVILEGE

Kernel-mode drivers	Drivers Code	Driver 1	Driver 2	Driver 3	OS Kernel
Driver 1	✓				
Driver 2		✓			
Driver 3			✓		
OS kernel	✓	✓	✓	✓	✓

<div>Slide 31 “Demo the Attack Prevention Final”</div> <div>The read and write access to the data is granted only to the drivers, which allocate these data buffers. All other memory access attempts are blocked.</div>	<div>MEMORY RANGER: PRINCIPLE OF LEAST PRIVILEGE</div> <table><thead><tr><th rowspan="2">Kernel-mode drivers</th><th colspan="4">Drivers Code</th><th colspan="3">Allocated Memory Data</th></tr><tr><th>Read</th><th>Write</th><th>Execute</th><th>Blocked</th><th>EPROCESS structures</th><th>Blocked</th></tr></thead><tbody><tr><td>Driver 1</td><td>✓</td><td>✓</td><td>✗</td><td>✗</td><td>✓</td><td>✗</td></tr><tr><td>Driver 2</td><td>✓</td><td>✓</td><td>✗</td><td>✗</td><td>✗</td><td>✓</td></tr><tr><td>Driver 3</td><td>✗</td><td>✓</td><td>✗</td><td>✗</td><td>✗</td><td>✓</td></tr><tr><td>OS kernel</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr></tbody></table>	Kernel-mode drivers	Drivers Code				Allocated Memory Data			Read	Write	Execute	Blocked	EPROCESS structures	Blocked	Driver 1	✓	✓	✗	✗	✓	✗	Driver 2	✓	✓	✗	✗	✗	✓	Driver 3	✗	✓	✗	✗	✗	✓	OS kernel	✓	✓	✓	✓	✓	✓
Kernel-mode drivers	Drivers Code				Allocated Memory Data																																						
	Read	Write	Execute	Blocked	EPROCESS structures	Blocked																																					
Driver 1	✓	✓	✗	✗	✓	✗																																					
Driver 2	✓	✓	✗	✗	✗	✓																																					
Driver 3	✗	✓	✗	✗	✗	✓																																					
OS kernel	✓	✓	✓	✓	✓	✓																																					
<div>Slide 32 “Demo the Attack Prevention Final”</div> <div>Memory Ranger protects drivers’ code from being dumped, it prevents read and write access to the allocated data, and finally, it prevents a privilege escalation attack. Let’s move on the Memory Ranger architecture.</div>	<div>MEMORY RANGER: PRINCIPLE OF LEAST PRIVILEGE</div> <table><thead><tr><th rowspan="2">Kernel-mode drivers</th><th colspan="4">Drivers Code</th><th colspan="3">Allocated Memory Data</th></tr><tr><th>Read</th><th>Write</th><th>Execute</th><th>Blocked</th><th>EPROCESS structures</th><th>Blocked</th></tr></thead><tbody><tr><td>Driver 1</td><td>✓</td><td>✗</td><td>✗</td><td>✗</td><td>✓</td><td>✗</td></tr><tr><td>Driver 2</td><td>✓</td><td>✗</td><td>✗</td><td>✗</td><td>✗</td><td>✓</td></tr><tr><td>Driver 3</td><td>✗</td><td>✓</td><td>✗</td><td>✗</td><td>✗</td><td>✓</td></tr><tr><td>OS kernel</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr></tbody></table>	Kernel-mode drivers	Drivers Code				Allocated Memory Data			Read	Write	Execute	Blocked	EPROCESS structures	Blocked	Driver 1	✓	✗	✗	✗	✓	✗	Driver 2	✓	✗	✗	✗	✗	✓	Driver 3	✗	✓	✗	✗	✗	✓	OS kernel	✓	✓	✓	✓	✓	✓
Kernel-mode drivers	Drivers Code				Allocated Memory Data																																						
	Read	Write	Execute	Blocked	EPROCESS structures	Blocked																																					
Driver 1	✓	✗	✗	✗	✓	✗																																					
Driver 2	✓	✗	✗	✗	✗	✓																																					
Driver 3	✗	✓	✗	✗	✗	✓																																					
OS kernel	✓	✓	✓	✓	✓	✓																																					
<div>Slide 33 “MemoryRanger architecture”</div> <div>This MemoryRanger includes the following three parts: MemoryMonRWX, DdiMon and Memory Access Policy. The first two parts were developed by Satoshi Tanda, during our collaboration. MemoryMonRWX processes the loading a new drivers and processes as well as all EPT violations, while DdiMon installs the invisible hooks on kernel functions, which allocate and deallocate /di:/ memory. I developed Memory Access Policy, which is a kind of brain for the first two parts.</div>	<div>MEMORY RANGER ARCHITECTURE: THREE PARTS</div> <div><div>Process is created</div><div>Driver is loaded</div><div>Memory is allocated</div><div>EPT violations: read, write, exec</div></div> <div><div>MemoryMonRWX*</div><div>DdiMon*</div><div>Memory Access Policy</div></div> <div></div> <div><small>* by Satoshi Tanda, @satanda, 1 https://github.com/satanda</small></div>																																										
<div>Slide 34 “MemoryRanger architecture”</div> <div>For each newly loaded process, MemoryRanger adds the following structure to the list. This structure includes process ID and the address of its EPROCESS structure.</div>	<div>MEMORY RANGER ARCHITECTURE: THREE PARTS</div> <div><div>Process is created</div><div>Driver is loaded</div><div>Memory is allocated</div><div>EPT violations: read, write, exec</div></div> <div><div>MemoryMonRWX*</div><div>DdiMon*</div><div>Memory Access Policy</div></div> <div><div>List of EPROCESSES Information</div></div> <div></div> <div><small>* by Satoshi Tanda, @satanda, 1 https://github.com/satanda</small></div>																																										
<div>Slide 35 “MemoryRanger architecture”</div> <div>For each newly loaded driver MemoryRanger adds the following structure to the list. This structure includes an address of allocated EPT structure for this driver, as well as drivers start and end addresses, and the vector to collect memory addresses, which will be allocated by this driver.</div>	<div>MEMORY RANGER ARCHITECTURE: THREE PARTS</div> <div><div>Process is created</div><div>Driver is loaded</div><div>Memory is allocated</div><div>EPT violations: read, write, exec</div></div> <div><div>MemoryMonRWX*</div><div>DdiMon*</div><div>Memory Access Policy</div></div> <div><div>List of EPROCESSES Information</div><div>List of Memory Enclaves</div></div> <div></div> <div><small>* by Satoshi Tanda, @satanda, 1 https://github.com/satanda</small></div>																																										
<div>Slide 36 “MemoryRanger architecture”</div> <div>Each time the driver allocates memory data MemoryRanger adds the following information to the list. This structure includes the address and size of newly allocated data.</div>	<div>MEMORY RANGER ARCHITECTURE: THREE PARTS</div> <div><div>Process is created</div><div>Driver is loaded</div><div>Memory is allocated</div><div>EPT violations: read, write, exec</div></div> <div><div>MemoryMonRWX*</div><div>DdiMon*</div><div>Memory Access Policy</div></div> <div><div>List of EPROCESSES Information</div><div>List of Memory Enclaves</div><div>List of Allocated Memory Pools</div></div> <div></div> <div><small>* by Satoshi Tanda, @satanda, 1 https://github.com/satanda</small></div>																																										

<p>Slide 37 “MemoryRanger architecture”</p> <p>Memory Access Policy manages all these lists and decides to grant memory access or to deny it.</p>	<p>MEMORY RANGER ARCHITECTURE: THREE PARTS</p>  <p><small>* by Behnke/Torvalds, @torvalds, & https://github.com/torvalds</small></p>
<p>Slide 38 “Memory Ranger Dispatcher”</p> <p>To put it in a nutshell the MemoryRanger dispatcher is here. It processes three EPT violations and one VM exit.</p>	<p>MEMORY RANGER DISPATCHER (SIMPLIFIED)</p> <pre> switch (exit_reason) { case (excrute_violation): change_epf(); break; case (read_violation write_violation): if (access_legal) == false: set_pte(pfn, read write, fake_page); set_monitor_trap_flag(); break; case (monitor_trap_flag): set_pte(pfn, no_access, original_page); clear_monitor_trap_flag(); break; } </pre>
<p>Slide 39 “Memory Ranger Dispatcher”</p> <p>For éxecúte violation MemoryRanger changes pointer to the appropriate EPT structure, so the kernel-mode code continues its execution into the corresponding EPT structure.</p>	<p>MEMORY RANGER DISPATCHER (SIMPLIFIED)</p> <pre> switch (exit_reason) { case (excrute_violation): change_epf(); break; case (read_violation write_violation): if (access_legal) == false: set_pte(pfn, read write, fake_page); set_monitor_trap_flag(); break; case (monitor_trap_flag): set_pte(pfn, no_access, original_page); clear_monitor_trap_flag(); break; } </pre>
<p>Slide 40 “Memory Ranger Dispatcher”</p> <p>☺ For read and write violations MemoryRanger checks whether this access is legal. For an illegal access, the MemoryRanger changes PFN value of the protected page to the fake one, and allows /ə'laʊz/ access to this fake data. Finally, it sets Monitor Trap Flag. As a result, after a driver reads the fake data the control goes to the hypervisor again.</p>	<p>MEMORY RANGER DISPATCHER (SIMPLIFIED)</p> <pre> switch (exit_reason) { case (excrute_violation): change_epf(); break; case (read_violation write_violation): if (access_legal) == false: set_pte(pfn, read write, fake_page); set_monitor_trap_flag(); break; case (monitor_trap_flag): set_pte(pfn, no_access, original_page); clear_monitor_trap_flag(); break; } </pre>
<p>Slide 41 “Memory Ranger Dispatcher”</p> <p>Now MemoryRanger restores PFN value to the protected page and blocks any access to this page. Finally, it resets the Monitor Trap Flag.</p>	<p>MEMORY RANGER DISPATCHER (SIMPLIFIED)</p> <pre> switch (exit_reason) { case (excrute_violation): change_epf(); break; case (read_violation write_violation): if (access_legal) == false: set_pte(pfn, read write, fake_page); set_monitor_trap_flag(); break; case (monitor_trap_flag): set_pte(pfn, no_access, original_page); clear_monitor_trap_flag(); break; } </pre>
<p>Slide 42 “How to your own data?”</p> <p>Here are the steps how to add a new data to be protected by my Memory Ranger.</p> <p>First of all you need to create a list of addresses and sizes of memory regions, which are needed to be protected and clear access bits to restrict the access. Do not forget to update this list, for example using callback function.</p>	<p>HOW TO PROTECT YOUR DATA IN MEMORY?</p> <ol style="list-style-type: none"> 1. Callback - creating a list of protected objects <ul style="list-style-type: none"> • Add objects' addresses & sizes to the list • Block memory access for objects memory via EPT 2. EPT dispatcher – process EPT violations for this data <ul style="list-style-type: none"> • type_of_access – read or write • guest_ip is the 'source address' • fault_va is the 'destination address' • Temp allow access to the data using MTF (page-align issue) • Redirect access to the fake data using MTF and EPT.PFN

<p>Slide 43 “How to your own data?”</p> <p>Then dispatch EPT violations, which occur during access to your data. To process all violations you have three input parameters: type of access, address of module, which tries to access and finally the address of data, which has been accessed. You can only monitor the access attempts as well as redirect them.</p>	<p>HOW TO PROTECT YOUR DATA IN MEMORY?</p> <ol style="list-style-type: none"> 1. Callback - creating a list of protected objects <ul style="list-style-type: none"> • Add objects' addresses & sizes to the list • Block memory access for objects memory via EPT 2. EPT dispatcher - process EPT violations for this data <ul style="list-style-type: none"> • type_of_access - read or write • guest_ip is the 'source address' • fault_va is the 'destination address' • Temp allow access to the data using MTF (page-align issue) • Redirect access to the false data using MTF and EPT.PFN 										
<p>Slide 44 “Memory Ranger benchmarks”</p> <p>Now let's move final step, which is the benchmark assessment.</p> <p>☺ I measure the time of legal access from driver to the allocated data in four situations. Without memory protection with enabled cache, with disabled cache, and with two memory protectors: AllMemPro and MemoryRanger. AllMemPro or Allocated Memory Protector is the closest competitor for MemoryRanger, which I developed a year ago. AllMemPro uses only one EPT structure to prevent illegal access to the allocated memory. I 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.</p>	<p>MEMORY RANGER BENCHMARKS: MEMORY ACCESS TIME</p>  <table border="1"> <thead> <tr> <th>Configuration</th> <th>Memory Access Time (ns)</th> </tr> </thead> <tbody> <tr> <td>Enabled Cache</td> <td>70±2</td> </tr> <tr> <td>Disabled Cache</td> <td>100,000±4,000</td> </tr> <tr> <td>AllMemPro</td> <td>800,000±10,000</td> </tr> <tr> <td>MemoryRanger</td> <td>170,000±7,000</td> </tr> </tbody> </table>	Configuration	Memory Access Time (ns)	Enabled Cache	70±2	Disabled Cache	100,000±4,000	AllMemPro	800,000±10,000	MemoryRanger	170,000±7,000
Configuration	Memory Access Time (ns)										
Enabled Cache	70±2										
Disabled Cache	100,000±4,000										
AllMemPro	800,000±10,000										
MemoryRanger	170,000±7,000										
<p>Slide 45 “Conclusion”</p> <p>We are close to the end of our dramatic story. Let me recap very briefly on what I have presented and what you learn. First of all, I've demonstrated that drivers can access kernel-mode memory without any security reaction. Then, I've presented MemoryRanger, which can prevent unauthorized memory access by running drivers in isolated enclaves. Finally, I'd like to highlight one <u>important thing</u>. MemoryRanger seems to prevent the recent side-channel attacks. If anyone has a proof-of-concepts of Meltdown or Spectre, please send it to me.</p>	<p>BLACK HAT SOUND BYTES OR CONCLUSION</p> <ul style="list-style-type: none"> • Kernel-mode memory is out of control • MemoryRanger isolates drivers execution by using a specific EPT structure for each driver • MemoryRanger seems to prevent Spectre and Meltdown CPU attacks: research is ongoing 										
<p>Slide 46 “divide et impera”</p> <p>I'd just like to conclude with old Latin expression. It is on the screen.</p> <p>I believe it is applicable not only for ancient Sparta but also for kernel-mode drivers in newest Windows.</p>	<p>Divide and Rule *</p> <p>from Latin divide et impera</p> <p><small>* Carlsberg, P. (2013). Spectre and Meltdown: A targeted latency 1900-1903 BC. Routledge.</small></p>										
<p>Slide 47 “Thank you”</p> <p>Thank you!</p>	<p>Thank you!</p> <p>Igor Korotkiy igor.korotkiy@gmail.com</p> <p>All the details & my CV are here igorkorotkiy.blogspot.com</p> <p>★ 🇺🇸 🇩🇪 🇬🇧 🇫🇷 🇮🇹</p>										