https://www.patreon.com/p_lkrg

LKRG
UNDER THE HOOD

Adam “pi3” Zabrocki

Private contact:

http://pi3.com.pl
pi3@pi3.com.pl
Twitter: @Adam_pi3

http://www.openwall.com/lkrg
Twitter: @Openwall

/USR/BIN/WHOAMI

- Adam ‘pi3’ Zabrocki

/USR/BIN/WHOAMI

- Adam ‘pi3’ Zabrocki

- Microsoft (currently)
- European Organization for Nuclear Research (CERN)
- Hispasec Sistemas
- Wroclaw Centre for Networking and Supercomputing
- Cigital

/USR/BIN/WHOAMI

- Adam ‘pi3’ Zabrocki

- Microsoft (currently)
- European Organization for Nuclear Research (CERN)
- Hispasec Sistemas
- Wroclaw Centre for Networking and Supercomputing
- Cigital

- Bughunting (Hyper-V, OpenSSH, gcc SSP/ProPolice, Apache, xpdf, more…) – CVE numbers
- Phrack magazine (Scraps of notes on remote stack overflow exploitation)
- The ERESI Reverse Engineering Software Interface

ACKNOWLEDGMENT

Alexander Peslyak (Александр Песляк)

a.k.a. Solar Designer

ACKNOWLEDGMENT

Alexander Peslyak (Александр Песляк)

a.k.a. Solar Designer

Special thanks to the following people for the constructive criticism and brainstorming in the past stages of the project development:

- Rafał "n3rgal" Wojtczuk
- Brad "spender" Spengler
- PaX Team… I mean "pipacs"

TABLE OF CONTENTS

WHAT IS LKRG?

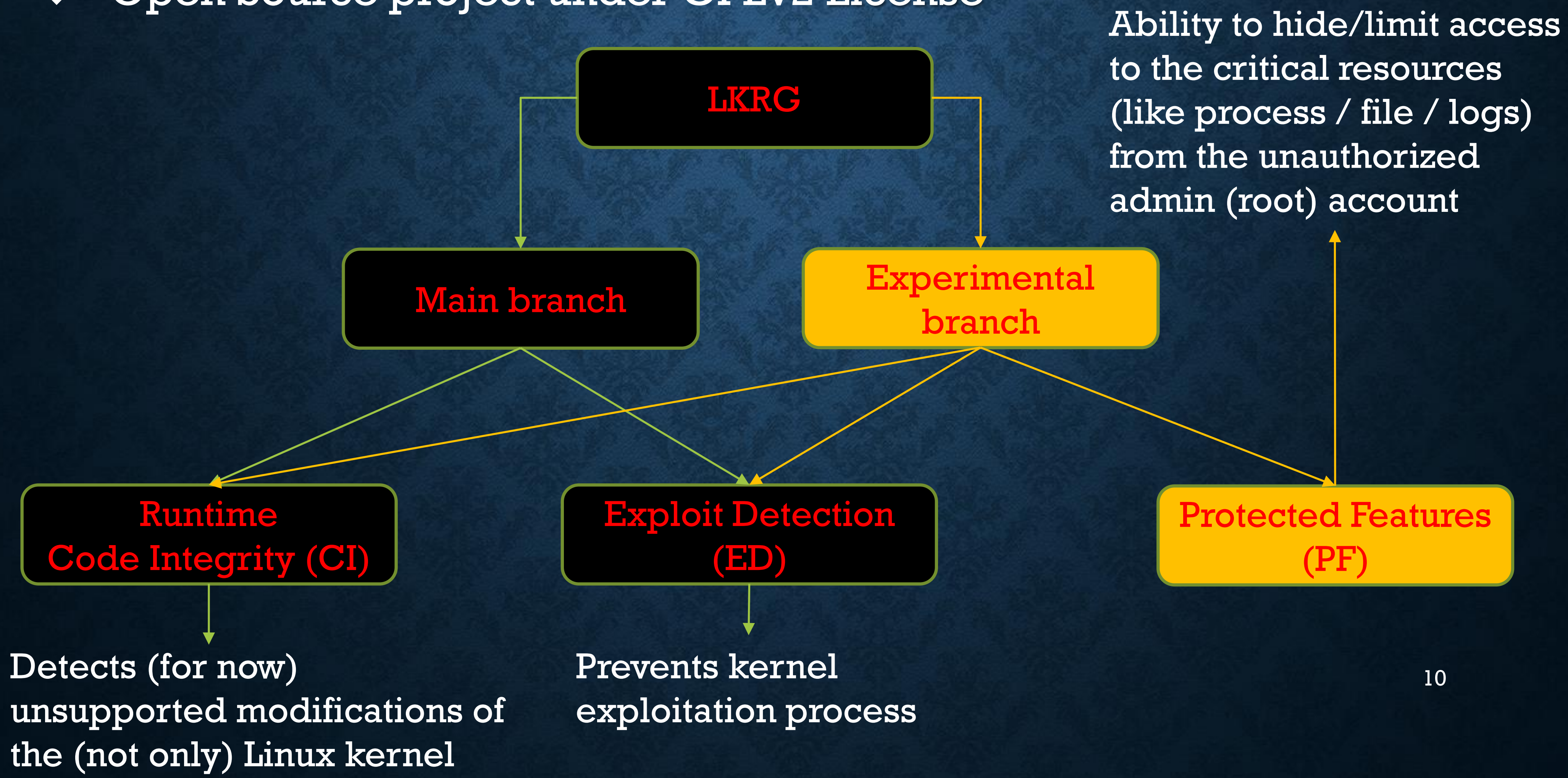
- LKRG – Linux Kernel Runtime Guard (self-explanatory ;p)

WHAT IS LKRG?

- LKRG – Linux Kernel Runtime Guard (self-explanatory ;p)
- Open Source project under GPLv2 License

WHAT IS LKRG?

- LKRG – Linux Kernel Runtime Guard (self-explanatory ;p)
- Open Source project under GPLv2 License



THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
  1. Attacking kernel from the boot chain
  2. Attacking kernel via kernel vulnerabilities
  3. Persistence of the attack e.g. kernel backdoors

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
  1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
  2. Attacking kernel via kernel vulnerabilities
  3. Persistence of the attack e.g. kernel backdoors

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
  1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
  2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
  3. Persistence of the attack e.g. kernel backdoors

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
    1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
    2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
    3. Persistence of the attack e.g. kernel backdoors <- Code Integrity

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
  1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
  2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
  3. Persistence of the attack e.g. kernel backdoors <- Code Integrity
  4. [*Experimental branch] Attacking user mode client:
     a. Attacking user mode process in running state
     b. Attacking user mode file on disk
     c. Attacking user mode process via raw memory access
     d. Attacking user mode file via raw disk access
     e. Intermediate attack for user mode process via attacking dependent code (e.g. shared libraries)

THREAT MODEL

❖ The following main attacking scenarios (buckets) can be described:

1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
3. Persistence of the attack e.g. kernel backdoors <- Code Integrity
4. [*Experimental branch] Attacking user mode client:
   a. Attacking user mode process in running state <- Protected Process
   b. Attacking user mode file on disk
   c. Attacking user mode process via raw memory access
   d. Attacking user mode file via raw disk access
   e. Intermediate attack for user mode process via attacking dependent code (e.g. shared libraries)

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
  1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
  2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
  3. Persistence of the attack e.g. kernel backdoors <- Code Integrity
  4. [*Experimental branch] Attacking user mode client:
     a. Attacking user mode process in running state <- Protected Process
     b. Attacking user mode file on disk <- Protected File
     c. Attacking user mode process via raw memory access
     d. Attacking user mode file via raw disk access
     e. Intermediate attack for user mode process via attacking dependent code (e.g. shared libraries)

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
  1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
  2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
  3. Persistence of the attack e.g. kernel backdoors <- Code Integrity
  4. [*Experimental branch] Attacking user mode client:
     a. Attacking user mode process in running state <- Protected Process
     b. Attacking user mode file on disk <- Protected File
     c. Attacking user mode process via raw memory access
     d. Attacking user mode file via raw disk access
     e. Intermediate attack for user mode process via attacking dependent code (e.g. shared libraries)

Virtually extended CAP_SYS_RAWIO

THREAT MODEL

- The following main attacking scenarios (buckets) can be described:
    1. ~~Attacking kernel from the boot chain~~ <- currently out-of-scope
    2. Attacking kernel via kernel vulnerabilities <- Exploit Detection
    3. Persistence of the attack e.g. kernel backdoors <- Code Integrity
    4. [*Experimental branch] Attacking user mode client:
        a. Attacking user mode process in running state <- Protected Process
        b. Attacking user mode file on disk <- Protected File
        c. Attacking user mode process via raw memory access
        d. Attacking user mode file via raw disk access
        e. Intermediate attack for user mode process via attacking dependent code (e.g. shared libraries) <- static binary + Protected File

Virtually extended CAP_SYS_RAWIO

EXPLOIT DETECTION

- The aim of it is to detect kernel exploitation process by detecting specific data corruption in the kernel.

EXPLOIT DETECTION

- The aim of it is to detect kernel exploitation process by detecting specific data corruption in the kernel.
- Current version of the feature maintains its own task list in the system and independently tracks critical attributes, including:
  - pointer value of the 'task_struct' itself
  - pid value
  - name of the process
  - pointer value of the 'cred' structure
  - pointer value of the 'real_cred' structure
  - UID / GID / EUID / EGID / SUID / SGID / FSUID / FSGID
  - SECCOMP:
    - TIF_SECCOMP flag
    - SECCOMP_FILTER_FLAG_TSYNC flag
    - mode
    - filters

EXPLOIT DETECTION

- The aim of it is to detect kernel exploitation process by detecting specific data corruption in the kernel.
- Current version of the feature maintains its own task list in the system and independently tracks critical attributes, including:
  - pointer value of the 'task_struct' itself
  - pid value
  - name of the process
  - pointer value of the 'cred' structure
  - pointer value of the 'real_cred' structure
  - UID / GID / EUID / EGID / SUID / SGID / FSUID / FSGID
  - SECCOMP:
    - TIF_SECCOMP flag
    - SECCOMP_FILTER_FLAG_TSYNC flag
    - mode
    - filters

Token/pointer swap attacks (illegal commit_creds())

EXPLOIT DETECTION

- The aim of it is to detect kernel exploitation process by detecting specific data corruption in the kernel.
- Current version of the feature maintains its own task list in the system and independently tracks critical attributes, including:
  - pointer value of the 'task_struct' itself
  - pid value
  - name of the process
  - pointer value of the 'cred' structure
  - pointer value of the 'real_cred' structure
  - UID / GID / EUID / EGID / SUID / SGID / FSUID / FSGID
  - SECCOMP:
    - TIF_SECCOMP flag
    - SECCOMP_FILTER_FLAG_TSYNC flag
    - mode
    - filters

Token/pointer swap attacks (illegal commit_creds())

Credential overwrite

EXPLOIT DETECTION

- The aim of it is to detect kernel exploitation process by detecting specific data corruption in the kernel.
- Current version of the feature maintains its own task list in the system and independently tracks critical attributes, including:
  - pointer value of the 'task_struct' itself
  - pid value
  - name of the process
  - pointer value of the 'cred' structure
  - pointer value of the 'real_cred' structure
  - UID / GID / EUID / EGID / SUID / SGID / FSUID / FSGID
  - SECCOMP:
    - TIF_SECCOMP flag
    - SECCOMP_FILTER_FLAG_TSYNC flag
    - mode
    - filters

Token/pointer swap attacks (illegal commit_creds())

Credential overwrite

Seccomp based sandbox escape

EXPLOIT DETECTION

- Additionally, LKRG is guarding the following SELinux variables:
  - selinux_enabled
  - selinux_enforcing

EXPLOIT DETECTION

- Additionally, LKRG is guarding the following SELinux variables:
  - selinux_enabled
  - selinux_enforcing
- The following values are also tracked but currently not used (but will be):
  - securebits
  - cap_inheritable
  - cap_permitted
  - cap_effective
  - cap_bset
  - cap_ambient
  - pointer value of the real user ID subscription
  - pointer value of the user namespace

EXPLOIT DETECTION

- Additionally, LKRG is guarding the following SELinux variables:
  - selinux_enabled
  - selinux_enforcing

  ← SELinux escape
- The following values are also tracked but currently not used (but will be):
  - securebits
  - cap_inheritable
  - cap_permitted
  - cap_effective
  - cap_bset
  - cap_ambient
  - pointer value of the real user ID subscription
  - pointer value of the user namespace

EXPLOIT DETECTION

- Additionally, LKRG is guarding the following SELinux variables:
  - selinux_enabled
  - selinux_enforcing

SELinux escape

- The following values are also tracked but currently not used (but will be):
  - securebits
  - cap_inheritable
  - cap_permitted
  - cap_effective
  - cap_bset
  - cap_ambient
  - pointer value of the real user ID subscription
  - pointer value of the user namespace

Capabilities based sandbox escape

EXPLOIT DETECTION

- Additionally, LKRG is guarding the following SELinux variables:
  - selinux_enabled
  - selinux_enforcing

  ← SELinux escape

- The following values are also tracked but currently not used (but will be):
  - securebits
  - cap_inheritable
  - cap_permitted
  - cap_effective
  - cap_bset
  - cap_ambient

  ← Capabilities based sandbox escape

  - pointer value of the real user ID subscription
  - pointer value of the user namespace

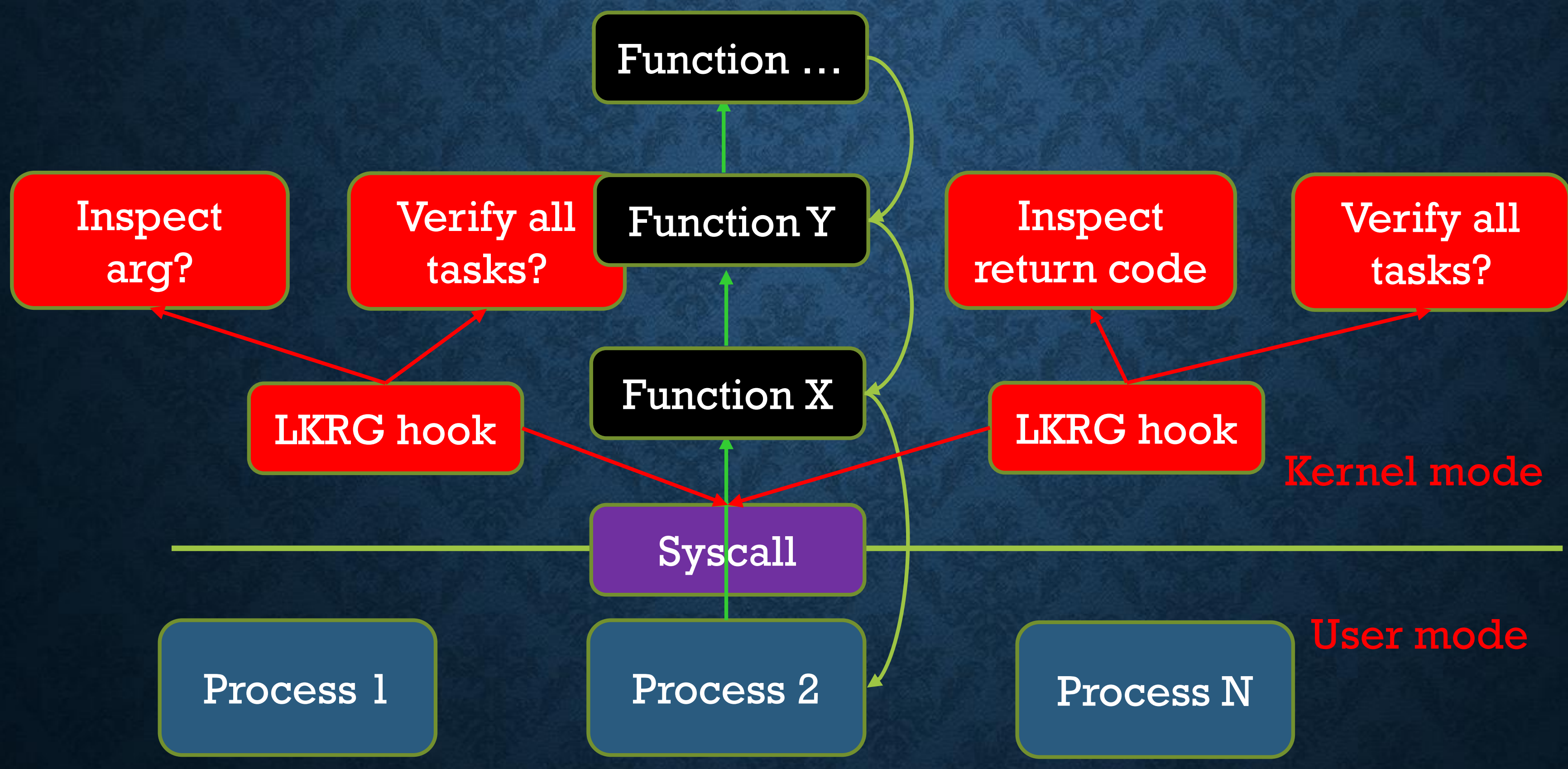
  ← Containers / namespace escape

EXPLOIT DETECTION

- How does LKRG build/maintain its own tasks list and update legit attributes changes?

EXPLOIT DETECTION

❖ How does LKRG build/maintain its own tasks list and update legit attributes changes?



EXPLOIT DETECTION

- How does LKRG build/maintain its own tasks list and update legit attributes changes?
- When does LKRG enforce integrity check?
  - setuid / setgid / seteuid / setegid / setreuid / setregid / setresuid / setresgid / setfsuid / setfsgid
  - setgroups
  - fork
  - execve
  - exit
  - do_init_module (covers init_module as well as finit_module)
  - delete_module
  - may_open (it is executed every time a user wants to open any resources in the system)
  - Whenever LKRG executes integrity checking function

EXPLOIT DETECTION

- How does LKRG build/maintain its own tasks list and update legit attributes changes?
- When does LKRG enforce integrity check?
  - setuid / setgid / seteuid / setegid / setreuid / setregid / setresuid / setresgid / setfsuid / setfsgid
  - setgroups
  - fork
  - execve
  - exit
  - do_init_module (covers init_module as well as finit_module)
  - delete_module
  - may_open (it is executed every time a user wants to open any resources in the system)
  - Whenever LKRG executes integrity checking function
- Checks are done for every process in the system, not just for the one which executed syscall (excluding may_open() for perf reasons). This list is not closed and will be evolving.

EXPLOIT DETECTION

Limitations

EXPLOIT DETECTION

- Limitations – "Bypassable" by design (for now) – difficult to protect from the same "trust level"

EXPLOIT DETECTION

- Limitations – "Bypassable" by design (for now) – difficult to protect from the same "trust level"
  - "Fly-under-LKRG's-radar"
  - Attack (disable) LKRG and continue normal work

EXPLOIT DETECTION

- Limitations – "Bypassable" by design (for now) – difficult to protect from the same "trust level"
  - "Fly-under-LKRG's-radar":
    - ✓ Overwrite critical metadata not guarded by LKRG
    - ✓ Relatively early-stage project – forgotten intercept?
    - ✓ Trying to win races (using not-intercepted syscalls)
    - ✓ "Move" attack to userspace
  - Attack (disable) LKRG and continue normal work

EXPLOIT DETECTION

- Limitations – "Bypassable" by design (for now) – difficult to protect from the same "trust level"
  - "Fly-under-LKRG's-radar":
    - ✓ Overwrite critical metadata not guarded by LKRG
    - ✓ Relatively early-stage project – forgotten intercept?
    - ✓ Trying to win races (using not-intercepted syscalls)
    - ✓ "Move" attack to userspace
  - Attack (disable) LKRG and continue normal work:
    - ✓ Trying to win races (corrupting LKRG's database)
    - ✓ Attack LKRG's internal synchronization / locking
    - ✓ Find all LKRG's running contexts and disable them + block a "new" one

EXPLOIT DETECTION

- Limitations – "Bypassable" by design (for now) – difficult to protect from the same "trust level"
  - "Fly-under-LKRG's-radar":
    - ✓ Overwrite critical metadata not guarded by LKRG
    - ✓ Relatively early-stage project – forgotten intercept?
    - ✓ Trying to win races (using not-intercepted syscalls)
    - ✓ "Move" attack to userspace
  - Attack (disable) LKRG and continue normal work:
    - ✓ Trying to win races (corrupting LKRG's database)
    - ✓ Attack LKRG's internal synchronization / locking
    - ✓ Find all LKRG's running contexts and disable them + block a "new" one
  - Directly attack the userspace via kernel (e.g. DirtyCOW)

EXPLOIT DETECTION

DEMO

RUNTIME CODE INTEGRITY

- Calculate hash from the critical [meta]data – SipHash

RUNTIME CODE INTEGRITY

- Calculate hash from the critical [meta]data – SipHash
- Guarded regions:
  - Critical (V)CPU/core data (currently only on x86/amd64 arch). Inter-Processor-Interrupt (IPI) is sent to individual core in all (V)CPUs to exclusively run LKRG function. Guard:

RUNTIME CODE INTEGRITY

- Calculate hash from the critical [meta]data – SipHash
- Guarded regions:
  - Critical (V)CPU/core data (currently only on x86/amd64 arch). Inter-Processor-Interrupt (IPI) is sent to individual core in all (V)CPUs to exclusively run LKRG function. Guard:
    - IDT entry point and size
    - IDT itself (as blob of memory)
    - MSRs:
      - MSR_IA32_SYSENTER_CS, MSR_IA32_SYSENTER_ESP, MSR_IA32_SYSENTER_EIP, MSR_IA32_CR_PAT, MSR_IA32_APICBASE, MSR_EFER, MSR_STAR, MSR_LSTAR, MSR_CSTAR, MSR_SYSCALL_MASK

RUNTIME CODE INTEGRITY

- Calculate hash from the critical [meta]data – SipHash
- Guarded regions:
  - Critical (V)CPU/core data (currently only on x86/amd64 arch). Inter-Processor-Interrupt (IPI) is sent to individual core in all (V)CPUs to exclusively run LKRG function. Guard:
    - IDT entry point and size
    - IDT itself (as blob of memory)
    - MSRs:
      - MSR_IA32_SYSENTER_CS, MSR_IA32_SYSENTER_ESP, MSR_IA32_SYSENTER_EIP, MSR_IA32_CR_PAT, MSR_IA32_APICBASE, MSR_EFER, MSR_STAR, MSR_LSTAR, MSR_CSTAR, MSR_SYSCALL_MASK
  - Additionally, LKRG keeps information about:
    - How many (V)CPUs/cores are available in the system
    - How many online (V)CPUs/cores are available in the system
    - How many offline (V)CPUs/cores are available in the system
    - How many possible (V)CPUs/cores might be available in the system

RUNTIME CODE INTEGRITY

- Guarded regions - continued:
  - Entire Linux kernel .text section
    - This covers almost entire Linux kernel itself, like syscall tables, all procedures, all function, all IRQ handlers, etc.
  - Linux kernel exception table
  - Entire Linux kernel .rodata section
  - Optionally IOMMU table
  - Modules

RUNTIME CODE INTEGRITY

- Guarded regions – continued – Modules:
  - For each individual module the following information is tracked based on module linked list:
    - Struct module pointer (a.k.a. THIS_MODULE)
    - Name
    - Pointer to the module_core
    - Size of the .text section
    - Hash from the entire .text section for that module

RUNTIME CODE INTEGRITY

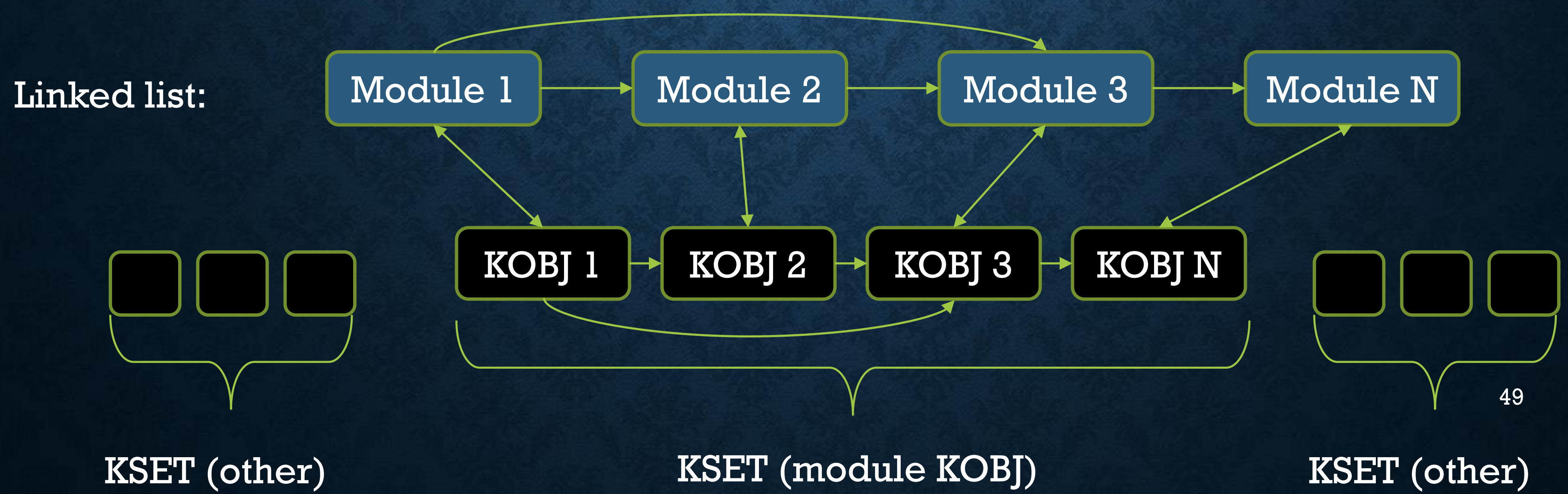
- Guarded regions – continued – Modules:
  - For each individual module the following information is tracked based on module linked list:
    - Struct module pointer (a.k.a. THIS_MODULE)
    - Name
    - Pointer to the module_core
    - Size of the .text section
    - Hash from the entire .text section for that module
  - For each individual module the following information is tracked based on KOBJs:
    - Struct module pointer (a.k.a. THIS_MODULE)
    - Pointer to the 'module_kobject' structure
    - Entire KOBJ structure (except from list_head and kref information)
    - Name
    - Pointer to the module_core
    - Size of the .text section
    - Hash from the entire .text section for that module

RUNTIME CODE INTEGRITY

- Guarded regions – continued – Modules:
  - Both pieces of information must match (if they exist in both places) and each of them is being tracked individually. Additionally, the following information is being tracked down:
    - Number of entries in module list
    - Number of KOBJs in specific KSET
    - Specific order of linked list in module list
    - Specific order in KSET for KOBJs

RUNTIME CODE INTEGRITY

- Guarded regions – continued – Modules:
  - Both pieces of information must match (if they exist in both places) and each of them is being tracked individually. Additionally, the following information is being tracked down:
    - Number of entries in module list
    - Number of KOBJs in specific KSET
    - Specific order of linked list in module list
    - Specific order in KSET for KOBJs



RUNTIME CODE INTEGRITY

- Guarded regions – continued – Modules:
  - Both pieces of information must match (if they exist in both places) and each of them is being tracked individually. Additionally, the following information is being tracked down:
    - Number of entries in module list
    - Number of KOBJs in specific KSET
    - Specific order of linked list in module list
    - Specific order in KSET for KOBJs
  - Dynamic module loading can be disabled via LKRG sysctl interface

RUNTIME CODE INTEGRITY

- When runtime CI validation routine is executed?
  - By the kernel timer interruption which generates work item and inserts it in shared WQ

RUNTIME CODE INTEGRITY

- When runtime CI validation routine is executed?
  - By the kernel timer interruption which generates work item and inserts it in shared WQ
  - On demand via a LKRG sysctl interface

RUNTIME CODE INTEGRITY

- When runtime CI validation routine is executed?
  - By the kernel timer interruption which generates work item and inserts it in shared WQ
  - On demand via a LKRG sysctl interface
  - Whenever any module activity is detected (e.g. loading / unloading)

RUNTIME CODE INTEGRITY

- When runtime CI validation routine is executed?
  - By the kernel timer interruption which generates work item and inserts it in shared WQ
  - On demand via a LKRG sysctl interface
  - Whenever any module activity is detected (e.g. loading / unloading)
  - Whenever a new (V)CPU or core activity is detected (hot CPU plug[in/off])

RUNTIME CODE INTEGRITY

- When runtime CI validation routine is executed?
  - By the kernel timer interruption which generates work item and inserts it in shared WQ
  - On demand via a LKRG sysctl interface
  - Whenever any module activity is detected (e.g. loading / unloading)
  - Whenever a new (V)CPU or core activity is detected (hot CPU plug[in/off])
  - On various random events in the system (see next slide)

RUNTIME CODE INTEGRITY

- The following events are monitored:
  - CPU idle – probability 0.005%
  - CPU frequency – probability 10%
  - CPU power management – probability 10%
  - Network device (e.g. device up/down) – probability 1%
  - Network event (e.g. ICMP redirects) – probability 5%
  - Network device IPv4 changes – probability 100%
  - Network device IPv6 changes – probability 100%
  - Task structure handing off – probability 0.01%
  - Task going out – probability 0.01%
  - Task calling do_munmap() – probability 0.005%
  - USB changes – probability 100%
  - Global AC events – probability 100%

RUNTIME CODE INTEGRITY

- Caveats:
  - *_JUMP_LABEL (self-modifying code)

RUNTIME CODE INTEGRITY

- Caveats:
  - *_JUMP_LABEL (self-modifying code)
    - If we detect that .text section for kernel was changed, we try to find the offset where modifications were made. We use this offset to calculate the VA of modified code. If modification happened because of the *_JUMP_LABEL options, either a long NOP or relative JMP instruction was injected (both are 5 bytes long):
      - If NOP is modified to JMP, destination of the instruction is still pointing to the inside of the same function (symbol name) where the modification happened. We decode this JMP instruction to validate if the target is still pointing inside the same symbol name range. If yes, it is most likely a 'legit' modification.
      - If JMP instruction was changed, we only allow it to be replaced by long NOP instruction.
    - Any other modifications are banned
    - More information can be found on the wiki page:

      http://openwall.info/wiki/p_lkrg/Main#JUMP_LABEL

RUNTIME CODE INTEGRITY

- Caveats:
  - *_JUMP_LABEL (self-modifying code)
    - If we detect that .text section for kernel was changed, we try to find the offset where modifications were made. We use this offset to calculate the VA of modified code. If modification happened because of the *_JUMP_LABEL options, either a long NOP or relative JMP instruction was injected (both are 5 bytes long):
      - If NOP is modified to JMP, destination of the instruction is still pointing to the inside of the same function (symbol name) where the modification happened. We decode this JMP instruction to validate if the target is still pointing inside the same symbol name range. If yes, it is most likely a 'legit' modification.
      - If JMP instruction was changed, we only allow it to be replaced by long NOP instruction.
    - Any other modifications are banned
    - More information can be found on the wiki page:

      http://openwall.info/wiki/p_lkrg/Main#JUMP_LABEL
    - Only for kernel core – not modules

RUNTIME CODE INTEGRITY

- Caveats:
  - IPI problem

RUNTIME CODE INTEGRITY

- Caveats:
  - IPI problem
    - There is an undesirable situation in SMP Linux machines while sending an IPI. Unfortunately, it might influence the state of the kernel and generate very confusing logs. They appear to suggest that the problem resides on the correct execution context which is killed and dumped, but not on the actually problematic context, which might not be dumped. This makes it hard to root-cause the problem even if one is aware of this shortcoming of the killings and the logging. More details about it can be found here:

      http://lists.openwall.net/linux-kernel/2016/09/21/68

COMMUNICATION CHANNEL

- Sysctl interface:

root@pi3-ubuntu:~/p_lkrg-main# sysctl -a | grep lkrg

lkrg.block_modules = 0

lkrg.clean_message = 1

lkrg.force_run = 0

lkrg.log_level = 1

lkrg.random_events = 1 (perf impact is around 2.5% for fully enabled LKRG, or around 0.7% for LKRG with code integrity checks on random events disabled)

lkrg.timestamp = 15

PERFORMANCE IMPACT

==================================

Project: john-1.8.0-jumbo-1

Configuration: ./configure CFLAGS='-O0'

Testing: make clean; time make -j 8

==================================

+02.513%

PERFORMANCE IMPACT

===================================

Project: john-1.8.0-jumbo-1

Configuration: ./configure CFLAGS='-O0'

Testing: make clean; time make -j 8

===================================

log_level=0, NO_EVENTS_CI		log_level=0, without_CI		log_level=0, Full LKRG	
real	+00.668%	real	+00.551%	real	+02.513%
user	-00.069%	user	-00.183%	user	-00.004%
sys	+07.200%	sys	+08.089%	sys	+08.355%

Full LKRG: ~2.5%

LKRG without random events: ~0.7%

LKRG IN RING -1

- Why not “ring -1” (hypervisor)?

LKRG IN RING -1

- Why not "ring -1" (hypervisor)?
  1. Some of the problems remain the same regardless of where the assist is implemented (ring 0 [kernel], ring -1 [hypervisor], ring -2 [SMM], ring -3 [AMT])

LKRG IN RING -1

- Why not "ring -1" (hypervisor)?
    1. Some of the problems remain the same regardless of where the assist is implemented (ring 0 [kernel], ring -1 [hypervisor], ring -2 [SMM], ring -3 [AMT])
    2. "Standarization" of hypervisor "world" in Linux is underdeveloped comparing to other modern OS (for now) – KVM? Xen? VirtualBox? Custom?

LKRG IN RING -1

- Why not "ring -1" (hypervisor)?
  1. Some of the problems remain the same regardless of where the assist is implemented (ring 0 [kernel], ring -1 [hypervisor], ring -2 [SMM], ring -3 [AMT])
  2. "Standarization" of hypervisor "world" in Linux is underdeveloped comparing to other modern OS (for now) – KVM? Xen? VirtualBox? Custom?
  3. "Closed" platforms don't have Linux-like problems – Samsung KNOX, Windows VSM, iOS KPP, etc.

LKRG IN RING -1

❖ Why not "ring -1" (hypervisor)?

1. Some of the problems remain the same regardless of where the assist is implemented (ring 0 [kernel], ring -1 [hypervisor], ring -2 [SMM], ring -3 [AMT])
2. "Standarization" of hypervisor "world" in Linux is underdeveloped comparing to other modern OS (for now) – KVM? Xen? VirtualBox? Custom?
3. "Closed" platforms don't have Linux-like problems – Samsung KNOX, Windows VSM, iOS KPP, etc.
4. Not all VPS solutions support nested virtualization – serious limitation

LKRG IN RING -1

- Why not "ring -1" (hypervisor)?
  1. Some of the problems remain the same regardless of where the assist is implemented (ring 0 [kernel], ring -1 [hypervisor], ring -2 [SMM], ring -3 [AMT])
  2. "Standarization" of hypervisor "world" in Linux is underdeveloped comparing to other modern OS (for now) – KVM? Xen? VirtualBox? Custom?
  3. "Closed" platforms don't have Linux-like problems – Samsung KNOX, Windows VSM, iOS KPP, etc.
  4. Not all VPS solutions support nested virtualization – serious limitation
  5. "ring -1" goes against mass deployment (same "kernel patch" solutions)

LKRG IN RING -1

- Why not "ring -1" (hypervisor)?
  1. Some of the problems remain the same regardless of where the assist is implemented (ring 0 [kernel], ring -1 [hypervisor], ring -2 [SMM], ring -3 [AMT])
  2. "Standarization" of hypervisor "world" in Linux is underdeveloped comparing to other modern OS (for now) – KVM? Xen? VirtualBox? Custom?
  3. "Closed" platforms don't have Linux-like problems – Samsung KNOX, Windows VSM, iOS KPP, etc.
  4. Not all VPS solutions support nested virtualization – serious limitation
  5. "ring -1" goes against mass deployment (same "kernel patch" solutions)
  6. Some of the servers / machines can't be rebooted (rebootless)

LKRG IN RING -1

When "Wild West of ring -1" becomes more unified, it'll be easy to add "ring -1" extension for LKRG which will guard "ring 0" instance. We will have 2 modes of operation: "weaker" without "ring -1" assist and stronger with hypervisor warranties – if environment supports it (still not the right time for it now!).

FUTURE

- Runtime Code Integrity:
- Exploit Detection:
- General:

FUTURE

- Runtime Code Integrity:
  - APIC / Local APIC
  - MADT / FADT / RSDT / ACPI
  - Call gates
  - Check if callbacks / notification routines point to the modules which we know and are tracking
- Exploit Detection:
- General:

FUTURE

- Runtime Code Integrity:
  - APIC / Local APIC
  - MADT / FADT / RSDT / ACPI
  - Call gates
  - Check if callbacks / notification routines point to the modules which we know and are tracking
- Exploit Detection:
  - Detect capabilities corruption
  - Detect containers / namespace escapes (Sandbox escapes)
  - Cover more kernel Elevation of Privileges (EoPs) techniques
- General:

FUTURE

- Runtime Code Integrity:
  - APIC / Local APIC
  - MADT / FADT / RSDT / ACPI
  - Call gates
  - Check if callbacks / notification routines point to the modules which we know and are tracking
- Exploit Detection:
  - Detect capabilities corruption
  - Detect containers / namespace escapes (Sandbox escapes)
  - Cover more kernel Elevation of Privileges (EoPs) techniques
- General:
  - Better self-defense:
    - Hash from the internal database
    - Hash from LKRG itself
  - Hypervisor extension (ring -1)
  - Probably more which I'm not aware of now :P

Private contact:

http://pi3.com.pl
pi3@pi3.com.pl
Twitter: @Adam_pi3

Q

and

A

http://www.openwall.com/lkrg
Twitter: @Openwall

Thanks and support LKRG! :)

https://www.patreon.com/p_lkrg