Linux Security Summit Europe 2018

Kernel Hardening:
Protecting the Protection Mechanisms

HUAWEI

Igor Stoppa - igor.stoppa@huawei.com
Cyber Security & Privacy Protection Labs - Huawei

- **introduction**
  - memory classification
  - memory protection mechanism
  - the plumbing
  - the porcelain
  - protection strategies
  - example of protection
  - limitations
  - future work
  - conclusion

Where it all begun - The Initial Problem

Scenario:

User of Android phone installs rogue App giving it every privilege.

Assumption:

The App is malicious and it ***will*** gain R/W access to kernel data.

Goal:

Prevent leak/loss of sensitive/critical information.

Reasoning

- Most attacks alter SELinux data.
- Write-Protect SELinux data to hinder the attack.
- Other subsystems can benefit from Write-protection

Primary use cases for memory protection

Accidental overwrites due to BUGs

Avoid corruption of (semi)constant data.

Malicious alterations

Prevent targeted alterations.

Accidental Overwrites due to BUGs

Not very demanding scenario, but still very useful

- Any coverage is better than nothing
- No special targets

Malicious Alterations

Most demanding scenario

- focused attacks
- "normal" corner cases become targets

Merging upstream: The quest for an example

Initial idea: SELinux policyDB & LSM Hooks

- **SELinux: complex data structures**
- **LSM Hooks: moving target**

Merging upstream: a better example

Thanks to Mimi Zohar: IMA list of measurements

- Simpler data structure than SELinux policyDB
- Less major changes than LSM Hooks
- Initial protection API was insufficient for the job

Learnings from protecting IMA measurements

Different write pattern from SELinux policyDB:

SELinux loads the DB after boot, then only reads it.
After writes subside, it can be protected.

Not a very common write pattern in kernel code.

Learnings from protecting IMA measurements

Different write pattern from SELinux policyDB:

IMA appends measurements indefinitely.
It must start protected and be modifiable later on

Fairly typical write pattern in kernel.

Need for "Write Rare" on dynamically allocated memory

What is the meaning of "Write Rare" memory?

- Primary mapping always R/O
- Means for controlled changes
- Acceptable overhead on write operations

"Acceptable" for use case & write pattern

Learnings from protecting IMA measurements

Statically allocated data:

- Protect also references to protected dynamic data
  Ex: head of a list
- Some references are written after kernel init

Need for "Write Rare" on statically allocated memory

Learnings from protecting IMA measurements

Often, the structures to protect belong to one or more lists.

- list node pointers must be protected as well
- other common data types require the same treatment

Need for a "Write Rare" variant of data structures

- **introduction**
- **memory classification**
- memory protection mechanism
- the plumbing
- the porcelain
- protection strategies
- example of protection
- limitations
- future work
- conclusion

Data memory: protections available

Statically allocated constant	Read Only (only after init)
Statically allocated variable	Read / Write
Statically allocated __ro_after_init	Writable only during kernel init
Dynamically allocated variable	Read / Write

Data memory: new protection proposed

Current moniker: **prmem**

(protected memory)

Statically allocated Write Rare after init **__wr_after_init**	● Read/Write during init ● Write Rare after init
Dynamically allocated **pmalloc**	● Read / Write till protected ● Write Rare or Read Only after protection ● From Write Rare to Read Only

Read Only vs Write Rare

Read Only

clean-cut transition, no need for a way back.

Write Rare (snake oil?)

- difficult to differentiate legitimate from rogue calls.
- one more obstacle during an attack

- **introduction**
- **memory classification**
- **memory protection mechanism**
- the plumbing
- the porcelain
- protection strategies
- example of protection
- limitations
- future work
- conclusion

Write Protection mechanism: the MMU

The MMU works at page level:

- SW must split allocations, based on writability
- An illegal write will trigger an Exception

- Two types of control and enforcement:
  - Kernel-Only
  - Kernel + [Hypervisor or TEE]

Kernel-exclusive control of the MMU

- the protection can be undone
- reduces the attack surface
- needs only a compliant MMU

Hypervisor-enforced memory protection

- Compromised kernel cannot undo the protection
- kernel can permanently relinquish capability
- Requires Hypervisor-capable HW

Some use-cases:

Big Irons: Large cloud providers, Data centers, etc.

Mobile devices: Samsung, Huawei, DarkMatter(?)

Regular distros: better protection, i.e. from containers

- **introduction**
- **memory classification**
- **memory protection mechanism**
- **the plumbing**
- the porcelain
- protection strategies
- example of protection
- limitations
- future work
- conclusion

prmem: requirements

- reads: negligible/acceptable overhead
- writes: acceptable overhead
- fallback to regular functions, if no MMU

prmem: pmalloc() implementation

- allocations grouped into logical pools
- built on top of vmalloc()
- more efficient with memory and TLB than vmalloc

prmem: __wr_after_init implementation

- Implementation and use similar to __ro_after_init
- Platform-dependent, due to mappings on arm64

prmem: write rare - Kernel-only - no Hypervisor

- Small pages, minimize exposure
- Disable local interrupts
- Random temporary R/W mapping for target page
- Inline functions, optimize away interfaces

prmem: write rare - with Hypervisor

- Hypervisor has own mappings, the kernel is irrelevant
- Hypervisor indifferent to kernel interrupts
- Inline functions, to reduce attack surface

prmem: plumbing components - status

Fully converted

- wr_memcpy()
- wr_memset()

Partially converted

- rcu:
    - wr_rcu_assign_pointer()
- atomic type:
    - wr_atomic_ulong

Side effects of the protection

- more targets for hardened-usercopy protection
- less vulnerable to use-after-free attacks and leaks
- Different use profile of the TLB

- **introduction**
- **memory classification**
- **memory protection mechanism**
- **the plumbing**
- **the porcelain**
- protection strategies
- example of protection
- limitations
- future work
- conclusion

Use of type aliasing

- Base type and its derived write-rare version
- Read operations on the base type
- Write operations on the write-rare version
- Use appropriate alignment for atomic operations
- No structure layout randomization

Example: aliasing of `struct hlist_node`

`struct hlist_node:`

`struct hlist_node *next`

`struct hlist_node **pprev`

`union prhlist_node:`

`union prhlist_node *next`

`union prhlist_node **pprev`

Aligned to sizeof(void *)

prmem: porcelain components - status

Fully converted

- list
- hlist
- list rcu
- hlist rcu

Pending

- object cache
- ...

- **introduction**
- **memory classification**
- **memory protection mechanism**
- **the plumbing**
- **the porcelain**
- **protection strategies**
- example of protection
- limitations
- future work
- conclusion

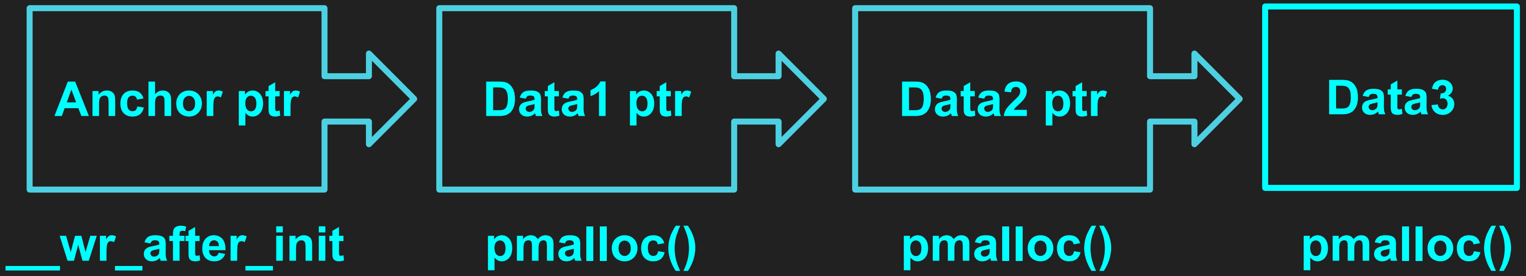
Protection patterns: anchored vs floating

Tradeoff between read and write overhead

Chained	Looped
No Read Overhead	Read Vetting Overhead
Data dependant Write Overhead	Data invariant Write Overhead

Chained protection: "chain of trust" of references

- easy way for converting existing code
- starts with the "anchor", an __[ro/wr]_after_init reference
- continues with a chain of links, similarly [ro/wr]
- no read overhead, but the "chain" can be very complex



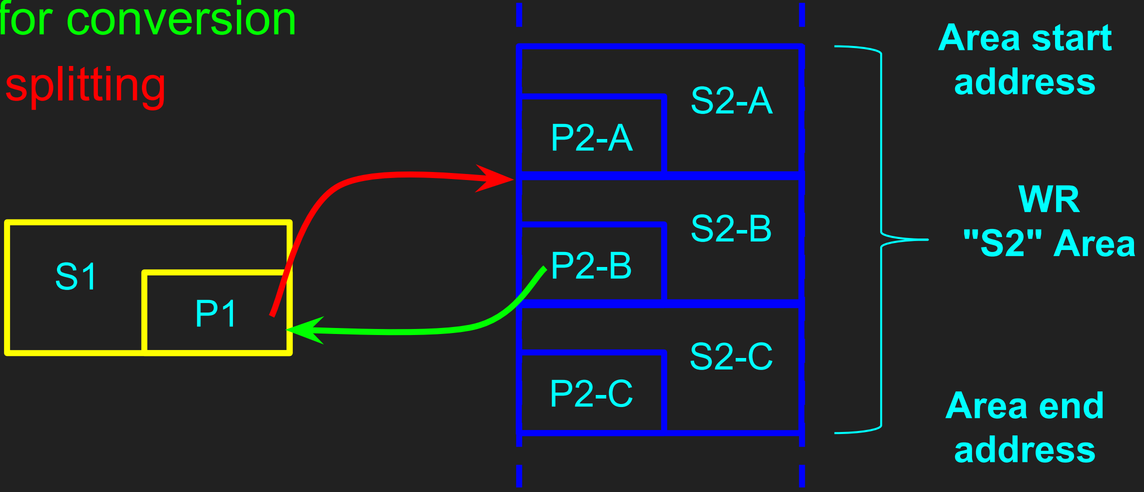
Looped protection: writable / write-rare ptr loop

[Credits: Samsung Knox - LSM protection]

- split R/W structure (s) into a writable one (s1) and a write-rare one (s2)
- writable structure (s1) has a pointer (p1) to a write-rare structure (s2)
- the write-rare structure (s2) has a pointer (p2) back to the writable pointer (p1)
- read overhead: before every use, vet p1, then verify the loop with p2
- simpler protection pattern for conversion
- existing structures require splitting

Vetting P1

- P1 within "S2" Area
- P1 aligned to S2 size
- P2 points to P1



- introduction
- memory classification
- memory protection mechanism
- the plumbing
- the porcelain
- protection strategies
- **example of protection**
- limitations
- future work
- conclusion

prmem example (no error-handling)

```
int *array __align(sizeof(void *)) __wr_after_init = NULL;
int size __wr_after_init = 0;
struct pmalloc_pool pool;

void alloc_array(void)
{
    int *p;

    pmalloc_init_pool(&pool, PMALLOC_MODE_WR);
    wr_int(&size, 5);  /* assignment */
    p = pcalloc(&pool, size, sizeof(int));
    wr_ptr(&array, &p);
}
```

Example: conversion of `struct hlist_node`

```
                                                         old code
                                                         new code
struct hlist_node {
union prhlist_node {          <- unnamed union, to obtain the punning
    struct hlist_node node __aligned(sizeof(void *));  <- base type
    struct { <- unnamed struct, for depth-compatibility with macros
    struct hlist_node *next;
        union prhlist_node *next __aligned(sizeof(void *));
    struct hlist_node **pprev;
        union prhlist_node **pprev __aligned(sizeof(void *));
    } __no_randomize_layout;              <- ensure consistent aliasing
};
} __aligned(sizeof(void *));             <- for atomic READ/WRITE on ptr
```

Example of conversion of a function

```
                                                    old code
                                                    new code
void hlist_del_init_rcu(struct hlist_node *n)
static __always_inline
void prhlist_del_init_rcu(union prhlist_node *n)
{
    if (!hlist_unhashed(n)) {
    if (!hlist_unhashed(&n->node)) { <-- reused R/O function
        __hlist_del(n);
        __prhlist_del(n); <--------------- drop-in W/R function
        n->pprev = NULL;
        prhlist_set_pprev(n, NULL); <---- replace assignment
    }
}
```

- **introduction**
- **memory classification**
- **memory protection mechanism**
- **the plumbing**
- **the porcelain**
- **protection strategies**
- **example of protection**
- **limitations**
- future work
- conclusion

Missing functionality

- __wr_after_init for arm64, without hypervisor
- no-MMU fallback for pmalloc() and wr()
- test cases for rcu operations
- test cases for atomic operations

Known vulnerabilities

- write rare API doesn't validate its caller
- pmalloc metadata is not write protected
- MMU page tables are exposed to rewrite attacks
- remapping depends on randomness of addresses

Performance limitation

Example: lists

1 list write operation -> multiple write rare operations

Each write rare operation has a cost:

- Kernel-only: handling new mapping
- Hypervisor: transitioning to hypervisor and back

- introduction
- memory classification
- memory protection mechanism
- the plumbing
- the porcelain
- protection strategies
- example of protection
- analysis
- present shortcomings
- **future work**
- conclusion

Add functionality

- create segment with mappings for __wr_after_init
- write fallback for no-MMU cases
- test cases for wr rcu and wr atomic
- basic hypervisor support (KVM)

prmem hardening

- vetting of call path to write rare
- research the protection of
  - pmalloc pool metadata
  - related vmalloc areas

prmem optimizations

- rewrite wr(h)list operations, if needed
    - less overhead
    - "data library" for hypervisor

More Kernel hardening

SELinux

- policyDB
- AVC
- containers

LSM Hooks

- containers

...

- introduction
- memory classification
- memory protection mechanism
- the plumbing
- the porcelain
- protection strategies
- example of protection
- present shortcomings
- future work
- **conclusion**

Does it work?

- Selected memory is write protected
- Procedure for converting existing code
- Simple overwrite attacks are harder to perform

Is it useful?

- Reduced attack surface
- Hypervisor can reduce it even more
- Not perfect: exposed to control flow attacks
- Opt-in protection, depending on the overhead

Thank You

References

- prmem patchset:
  https://github.com/Igor-security/linux/tree/wip

- Huawei kernel with early SELinux policyDB & LSM protection:
  (from tarball available from Huawei website, see README)
  https://github.com/Igor-security/Huawei_NEO

- Samsung kernel with Knox:
  (from tarball available from Samsung website, see README)
  https://github.com/Igor-security/Samsung_SM-N960F