Kernel-Hack-Drill Masterclass

Alexander Popov

positive technologies

May 5, 2026

Who Am I

- Alexander Popov
- Linux kernel developer since 2012
- Maintainer of some free software projects
- Principal Security Researcher and Head of Open Source Program Office at positive technologies

- Conference speaker:

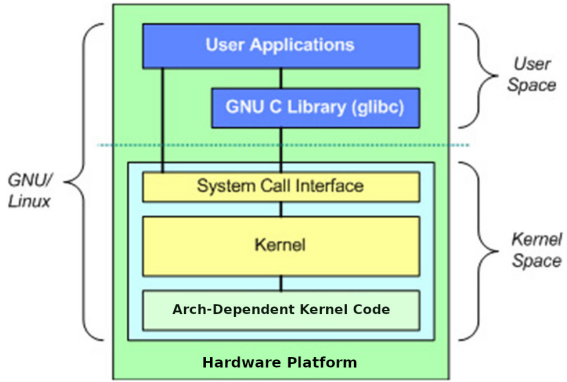
  Zer0Con, OffensiveCon, H2HC, Nullcon Goa, Linux Security Summit, Still Hacking Anyway, HITB, Positive Hack Days, ZeroNights, HighLoad++, Open Source Summit, OS Day, Linux Plumbers...

  a13xp0p0v.github.io/conference_talks

Agenda

1. Quick intro into Linux kernel security
2. Story behind **kernel-hack-drill**
3. Technical details
4. Plenty of live demos

OS Architecture (GNU/Linux)



ibm.com/developerworks/linux/library/l-linux-kernel/

OS Security Model

- An OS threat model may include
  1. Parsing and handling untrusted data
  2. Executing untrusted applications (userspace code)
  3. Communicating via untrusted networks

Yefim Deshalyt: The heroic defense of Old Ryazan

OS Security Model

- An OS threat model may include
  1. Parsing and handling untrusted data
  2. Executing untrusted applications (userspace code)
  3. Communicating via untrusted networks
- These attack vectors against OS may be used for
  1. Remote code execution (RCE)
  2. Local privilege escalation (LPE)
  3. Denial of service (DoS)
  4. Data leaks

Yefim Deshalyt: The heroic defense of Old Ryazan

OS Security Model

- An OS threat model may include
  1. Parsing and handling untrusted data
  2. Executing untrusted applications (userspace code)
  3. Communicating via untrusted networks
- These attack vectors against OS may be used for
  1. Remote code execution (RCE)
  2. Local privilege escalation (LPE)
  3. Denial of service (DoS)
  4. Data leaks

Yefim Deshalyt: The heroic defense of Old Ryazan

- An OS security model defines how OS security features mitigate these threats
- Excellent example: Android Platform Security Model

Linux Kernel Security

Linux kernel security is a very complex knowledge area.
Key concepts:

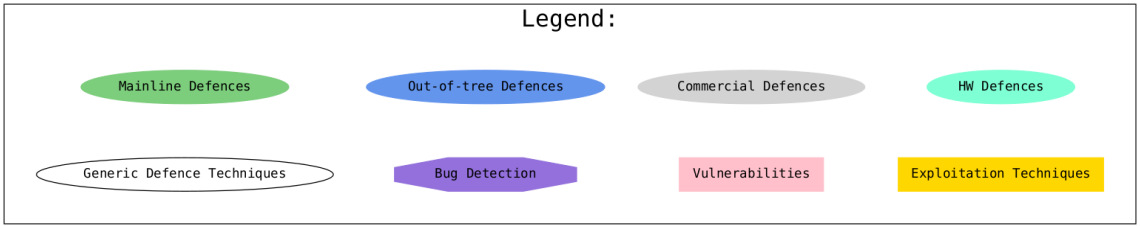
1. Vulnerability classes
2. Exploitation techniques
3. Bug detection mechanisms
4. Defence technologies
   - In the mainline
   - Shipped separately (commercial or under development)
   - Requiring special hardware features

All they have complicated relations with each other...
It would be nice to have a graphical representation of it.

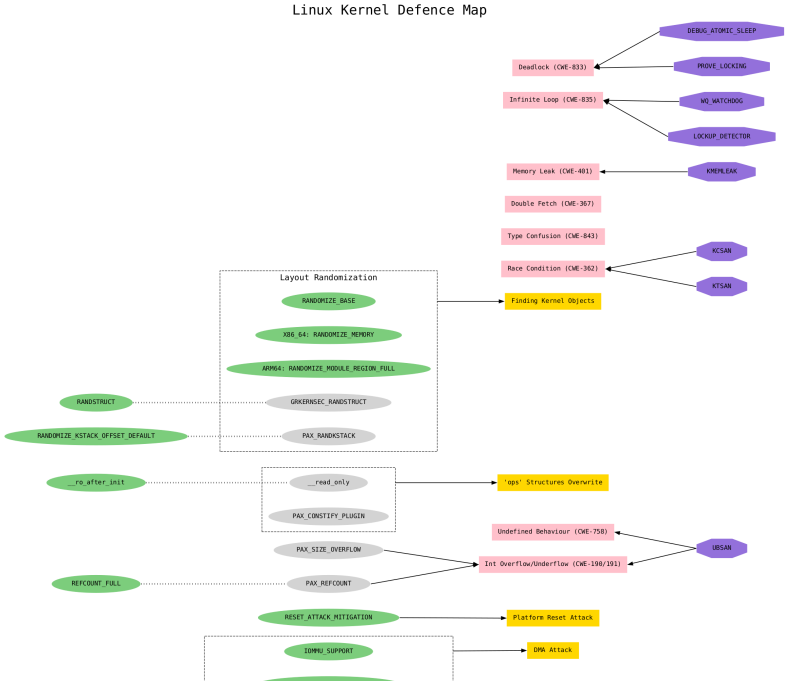
Drawn by Daniel Reeve, made by weta

Linux Kernel Defence Map

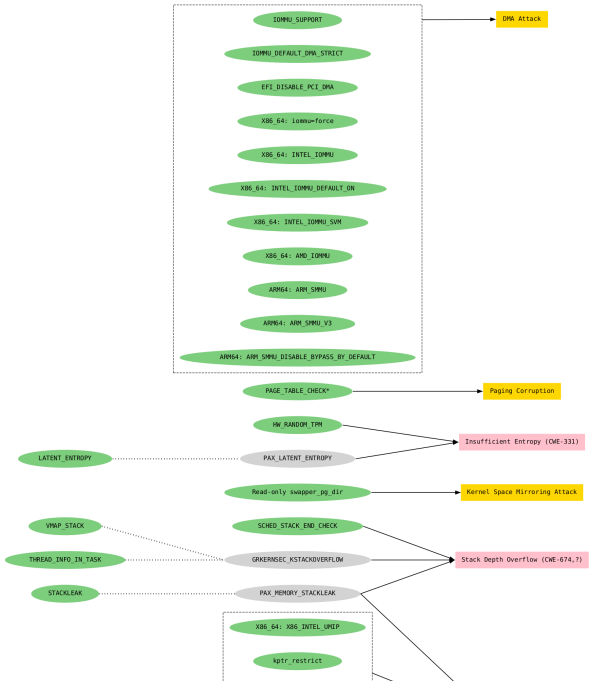
- So I developed the **Linux Kernel Defence Map**
  github.com/a13xp0p0v/linux-kernel-defence-map
- I started this project in 2018, and I'm continuing to improve and update it
- Map legend with key concepts:



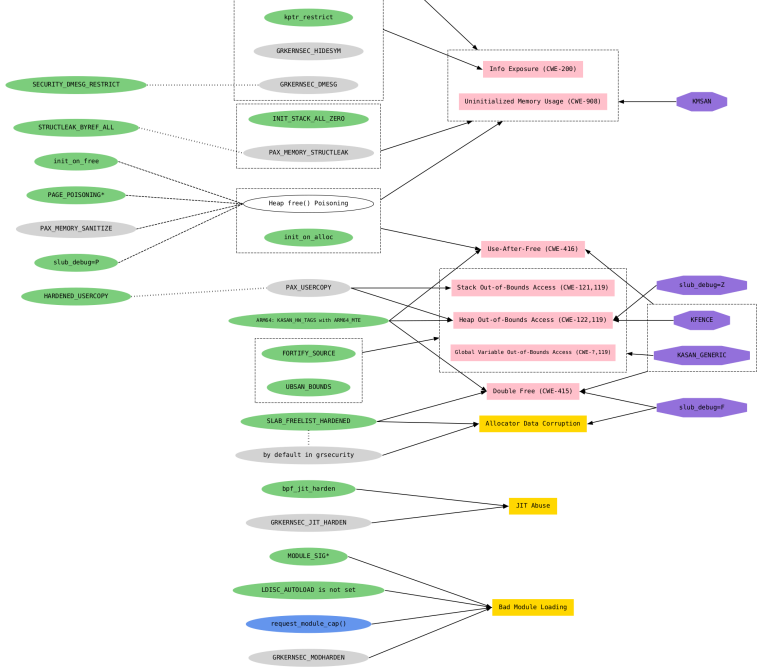
Linux Kernel Defence Map, whole picture (1/6)



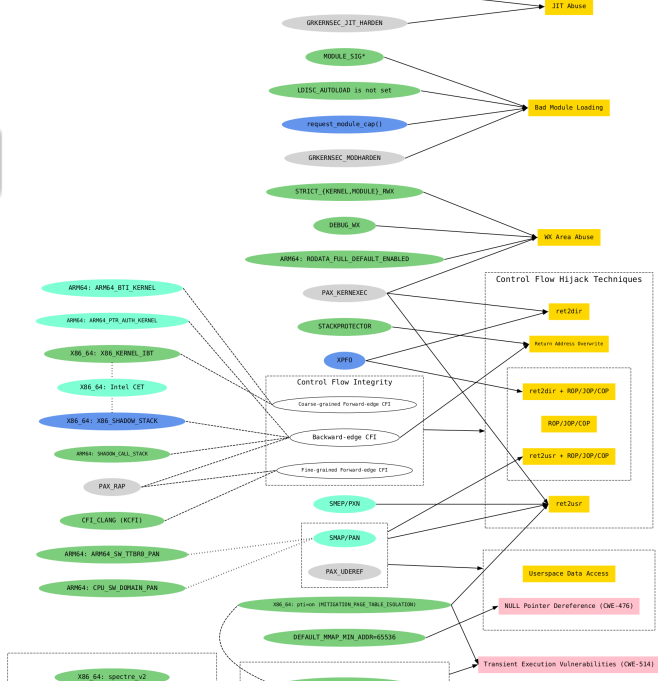
Linux Kernel Defence Map, whole picture (2/6)



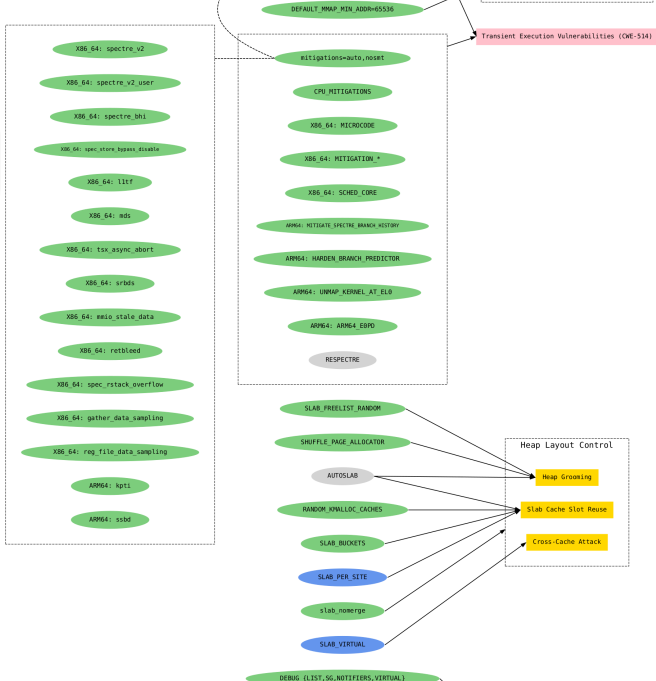
Linux Kernel Defence Map, whole picture (3/6)



Linux Kernel Defence Map, whole picture (4/6)



Linux Kernel Defence Map, whole picture (5/6)



Linux Kernel Defence Map, whole picture (6/6)

Linux Kernel Defence Map: How It Works

- Each connection between nodes represents some kind of relationship between them
- The node connections don't always mean "full mitigation"
- The map helps with navigating documentation and Linux kernel sources

Linux Kernel Defence Map: How It Works

- Each connection between nodes represents some kind of relationship between them
- The node connections don't always mean "full mitigation"
- The map helps with navigating documentation and Linux kernel sources
- The map provides the Common Weakness Enumeration (CWE) numbers for vuln classes

Linux Kernel Defence Map: How It Works

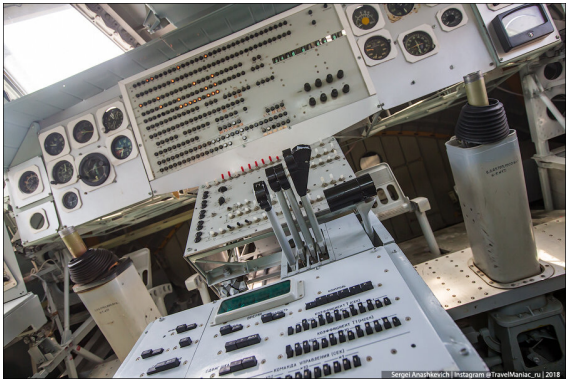
- Each connection between nodes represents some kind of relationship between them
- The node connections don't always mean "full mitigation"
- The map helps with navigating documentation and Linux kernel sources
- The map provides the Common Weakness Enumeration (CWE) numbers for vuln classes
- This map describes kernel security hardening
- [!] The map doesn't cover
  - Cutting the attack surface
  - Userspace security features
  - Security policies enforced by Linux Security Modules (LSM)

Nice Map! But What's in Practice?

Victor Belov: Soviet scientists theorists (1972)

Linux Kernel Parameters

- Kconfig options (compile-time)
- Kernel cmdline arguments (boot-time)
- Sysctl parameters (runtime)



Buran spacecraft control panel

Defender's Perspective

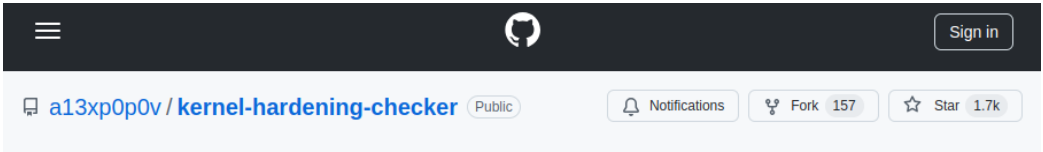
- There are **plenty** of Linux kernel security parameters
- A lot of them are **not enabled** by the major distros
- **Nobody** likes checking configs manually

Defender's Perspective

- There are **plenty** of Linux kernel security parameters
- A lot of them are **not enabled** by the major distros
- **Nobody** likes checking configs manually
- **So let computers do their job!**

Defender's Perspective

- There are **plenty** of Linux kernel security parameters
- A lot of them are **not enabled** by the major distros
- **Nobody** likes checking configs manually
- **So let computers do their job!**
- I created **kernel-hardening-checker** for checking the security-related parameters of the Linux kernel: github.com/a13xp0p0v/kernel-hardening-checker
- I started this work in 2018, the project is in continuous development



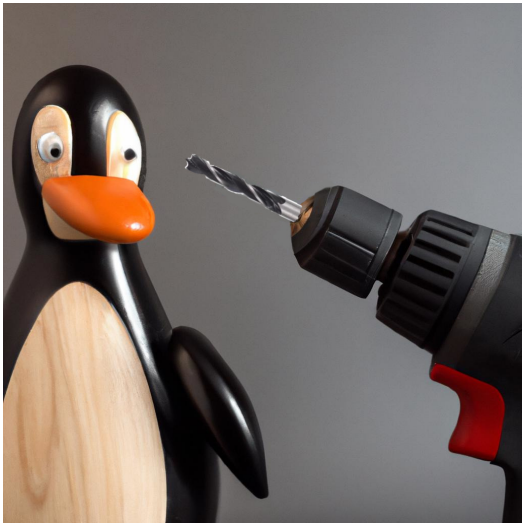
kernel-hardening-checker

Attacker's Perspective

- Back in 2017, I created a pet project for my students called **kernel-hack-drill**
- It provides a testing environment for learning Linux kernel security
- Then this spare-time project hadn't seen an update in years
- I revisited **kernel-hack-drill** during my **CVE-2024-50264** research:
  a13xp0p0v.github.io/2025/09/02/kernel-hack-drill-and-CVE-2024-50264.html
- I used it as a **testing ground** for developing the exploit primitives
  for this **extremely hard** race condition
- Now this project offers a solid set of resources for security researchers

Kernel Hack Drill: Fun Fact

Kernel Hack Drill: Fun Fact



Kernel Hack Drill

- Open-source project: github.com/a13xp0p0v/kernel-hack-drill
- Provides test environment for developing the Linux kernel exploit primitives you need
- Includes a good step-by-step setup guide in the README (kudos to the contributors!)
- A bit similar to github.com/hacktivesec/KRWX, but
  - Much simpler
  - Contains interesting PoC exploits

https://www.pngall.com/wp-content/uploads/4/Drill-Machine-PNG-Free-Download.png

Kernel Hack Drill Contents: Kernel Module

1. `drill_mod.c`
   - A small Linux kernel module
   - Provides `/proc/drill_act` file as a simple interface to userspace
   - Contains nice vulnerabilities that you control
2. `drill.h`
   - Header file describing the `drill_mod.ko` interface
3. `drill_test.c`
   - Userspace test for `drill_mod.ko`
   - It also passes if `CONFIG_KASAN=y`

```
#define DRILL_N 10240
#define DRILL_ITEM_SIZE 95
struct drill_item_t {
    unsigned long foobar;
    void (*callback)(void);
    char data[]; /* C99 flexible array */
};
enum drill_act_t {
    DRILL_ACT_NONE = 0,
    DRILL_ACT_ALLOC = 1,
    DRILL_ACT_CALLBACK = 2,
    DRILL_ACT_SAVE_VAL = 3,
    DRILL_ACT_FREE = 4,
    DRILL_ACT_RESET = 5
};
```

Kernel Hack Drill Contents: PoC Exploits (Part I)

1. `drill_uaf_callback.c`
   - UAF exploit invoking a callback in the freed `drill_item_t` struct
   - It performs control flow hijack and gains LPE

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part I)

1. `drill_uaf_callback.c`
   - UAF exploit invoking a callback in the freed `drill_item_t` struct
   - It performs control flow hijack and gains LPE
2. `drill_uaf_w_msg_msg.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - It performs a cross-cache attack, overwrites `msg_msg.m_ts`
   - It enables out-of-bounds read of the kernel memory

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part I)

1. `drill_uaf_callback.c`
   - UAF exploit invoking a callback in the freed `drill_item_t` struct
   - It performs control flow hijack and gains LPE

2. `drill_uaf_w_msg_msg.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - It performs a cross-cache attack, overwrites `msg_msg.m_ts`
   - It enables out-of-bounds read of the kernel memory

3. `drill_uaf_w_pipe_buffer.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - It performs cross-cache attack, overwrites `pipe_buffer.flags`
   - It implements the Dirty Pipe attack and gains LPE

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part II)

In collaboration with @d1sgr4c3 (thanks for the contribution!)

4. `drill_uaf_callback_rop_smep.c`
   - An improved version of `drill_uaf_callback.c`
   - It adds a ROP chain in userspace
   - That allows to bypass SMEP and page table isolation on x86_64

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part II)

In collaboration with @d1sgr4c3 (thanks for the contribution!)

4. `drill_uaf_callback_rop_smep.c`
   - An improved version of `drill_uaf_callback.c`
   - It adds a ROP chain in userspace
   - That allows to bypass SMEP and page table isolation on x86_64

5. `drill_uaf_callback_rop_smap.c`
   - An improved version of `drill_uaf_callback_rop_smep.c`
   - It places the ROP chain in the kernelspace to also bypass SMAP on x86_64

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part III)

In collaboration with @Willenst (thanks for the contribution!)

6. `drill_uaf_w_pte.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - Performs a cross-allocator attack
   - Overwrites `Page Table Entry (PTE)`
   - Implements the Dirty Pagetable attack and gains LPE

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part III)

In collaboration with @Willenst (thanks for the contribution!)

6. `drill_uaf_w_pte.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - Performs a cross-allocator attack
   - Overwrites `Page Table Entry (PTE)`
   - Implements the Dirty Pagetable attack and gains LPE

7. `drill_uaf_w_pud.c`
   - UAF exploit writing data to the freed `drill_item_t` struct
   - Performs cross-allocator attack
   - Overwrites `Page Upper Directory (PUD)`
   - Implements the Dirty Pagetable attack via huge pages (LPE)

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part IV)

In collaboration with @d1sgr4c3 (thanks for the contribution!)

8. `drill_oob_w_pipe_buffer.c`
   - a basic OOBW exploit
   - Corrupts the `pipe_buffer.page` pointer
   - Gains AARW of kernel memory via a pipe
   - Performs LPE

https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill Contents: PoC Exploits (Part IV)

In collaboration with @d1sgr4c3 (thanks for the contribution!)

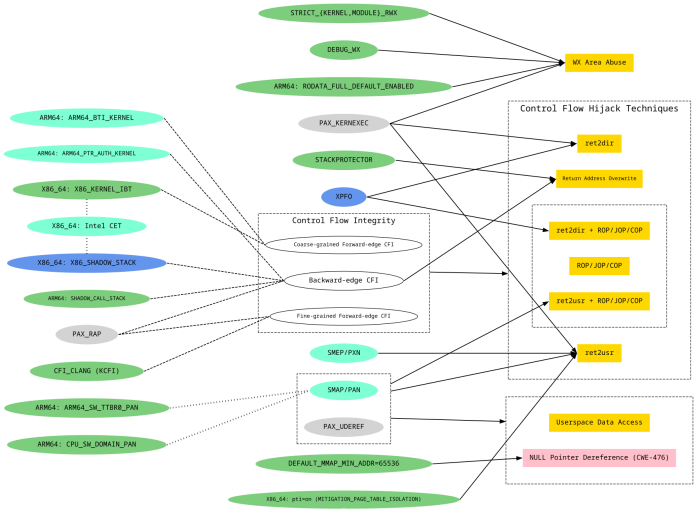
8. `drill_oob_w_pipe_buffer.c`
   - a basic OOBW exploit
   - Corrupts the `pipe_buffer.page` pointer
   - Gains AARW of kernel memory via a pipe
   - Performs LPE

9. More PoC exploits will come soon!

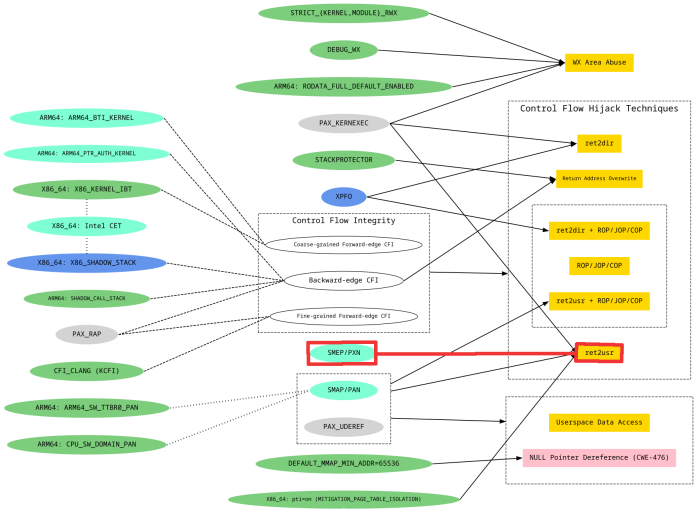
https://www.printables.com/model/78077-drill-guide

Kernel Hack Drill: Control Flow Hijacking

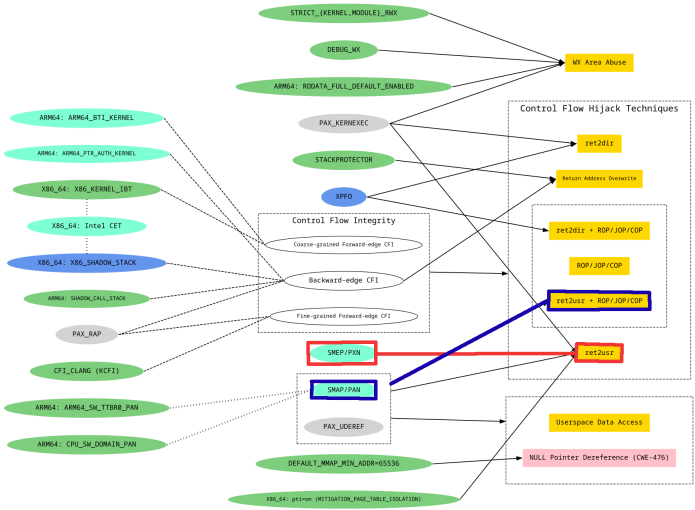
Example from the Map: Control-Flow Hijack



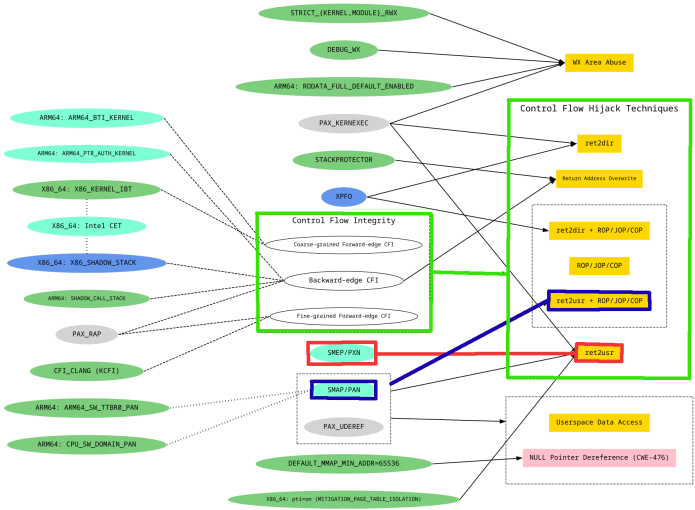
Example from the Map: Control-Flow Hijack



Example from the Map: Control-Flow Hijack

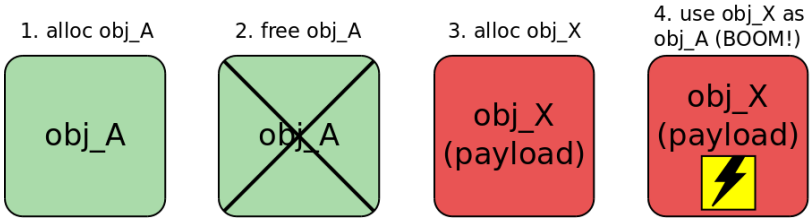


Example from the Map: Control-Flow Hijack



UAF Exploitation

- Naive UAF exploitation:

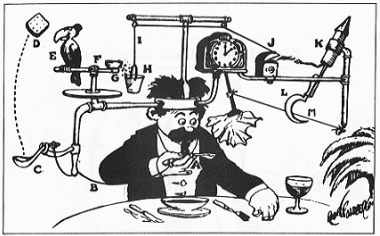


- But! Modern Linux kernel has slab allocator hardening features:
  - `CONFIG_RANDOM_KMALLOC_CACHES=y`
  - `CONFIG_SLAB_BUCKETS=y`
- Here cross-cache attack comes to play

Cross-Cache Attack in Kernel Hack Drill (Part 1)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

Before the attack:



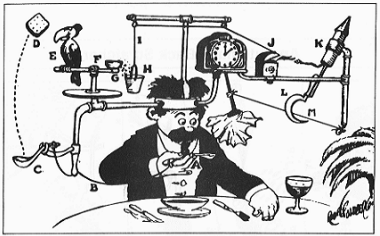
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 1)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

Before the attack:

1. Get the number of objects in the initial slab (containing a vulnerable object):
   see `objs_per_slab` in `/sys/kernel/slab/`



en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 1)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

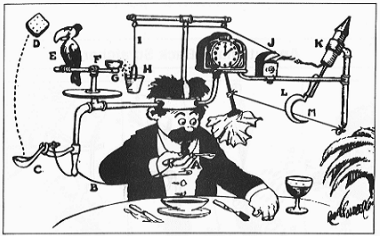
Before the attack:

1. Get the number of objects in the initial slab (containing a vulnerable object):

   see `objs_per_slab` in `/sys/kernel/slab/`

2. Get the number of per-CPU partial slabs in the initial `kmem_cache`:

   see `kmem_cache.cpu_partial_slabs` in gdb



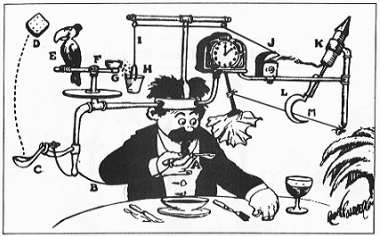
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 1)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

Before the attack:

1. Get the number of objects in the initial slab (containing a vulnerable object):
   see `objs_per_slab` in `/sys/kernel/slab/`
2. Get the number of per-CPU partial slabs in the initial `kmem_cache`:
   see `kmem_cache.cpu_partial_slabs` in gdb
3. Get the minimum number of per-node partial slabs in the initial `kmem_cache`: see `min_partial` in `/sys/kernel/slab/`



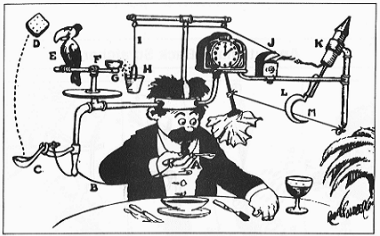
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 1)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

Before the attack:

1. Get the number of objects in the initial slab (containing a vulnerable object):
   see `objs_per_slab` in `/sys/kernel/slab/`
2. Get the number of per-CPU partial slabs in the initial `kmem_cache`:
   see `kmem_cache.cpu_partial_slabs` in gdb
3. Get the minimum number of per-node partial slabs in the initial `kmem_cache`: see `min_partial` in `/sys/kernel/slab/`
4. Ensure that `min_partial` $<$ `cpu_partial_slabs`, otherwise the described attack strategy will not work



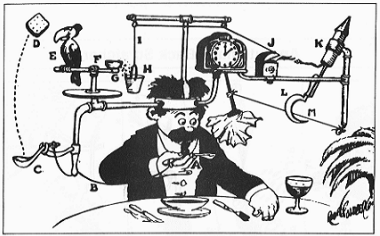
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 1)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

Before the attack:

1. Get the number of objects in the initial slab (containing a vulnerable object): see `objs_per_slab` in `/sys/kernel/slab/`
2. Get the number of per-CPU partial slabs in the initial `kmem_cache`: see `kmem_cache.cpu_partial_slabs` in gdb
3. Get the minimum number of per-node partial slabs in the initial `kmem_cache`: see `min_partial` in `/sys/kernel/slab/`
4. Ensure that `min_partial` < `cpu_partial_slabs`, otherwise the described attack strategy will not work
5. Estimate the number of holes in the initial and final slab caches

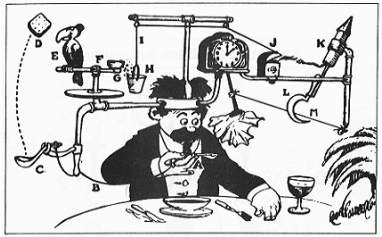


en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:



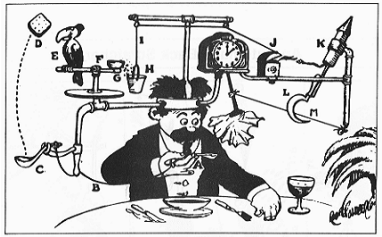
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:

1. Plug the holes in the initial and final slab caches



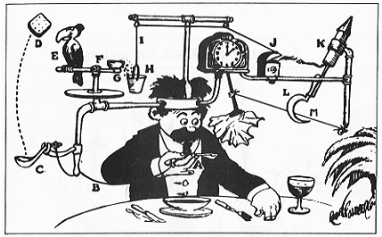
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)



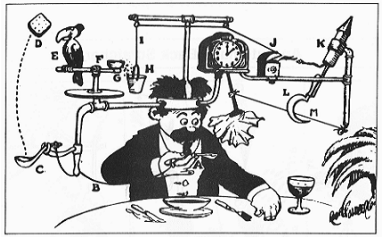
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)
3. Create a new active slab: allocate `objs_per_slab` objects



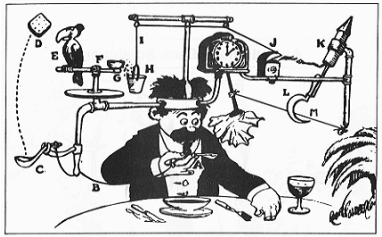
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)
3. Create a new active slab: allocate `objs_per_slab` objects
4. Allocate a vulnerable object



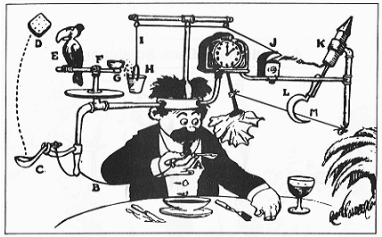
en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)
3. Create a new active slab: allocate `objs_per_slab` objects
4. Allocate a vulnerable object
5. Create a new active slab again: allocate `objs_per_slab` objects

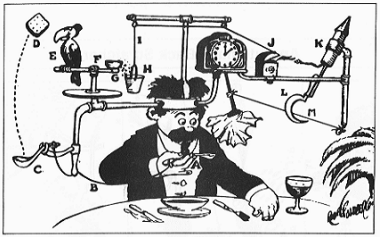


en.wikipedia.org/wiki/Rube_Goldberg_machine

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:



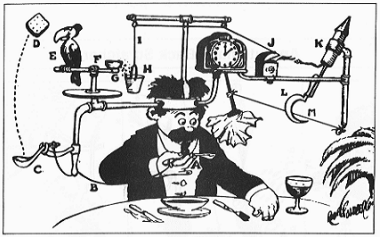
en.wikipedia.org/wiki/Rube_Goldberg_machine

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)
3. Create a new active slab: allocate `objs_per_slab` objects
4. Allocate a vulnerable object
5. Create a new active slab again: allocate `objs_per_slab` objects
6. Free the slab with the UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:



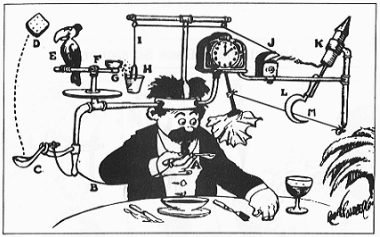
en.wikipedia.org/wiki/Rube_Goldberg_machine

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)
3. Create a new active slab: allocate `objs_per_slab` objects
4. Allocate a vulnerable object
5. Create a new active slab again: allocate `objs_per_slab` objects
6. Free the slab with the UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one
7. Fill up the partial list: free one of each `objs_per_slab` objects in the reserved slabs

Cross-Cache Attack in Kernel Hack Drill (Part 2)

See the full details in the code: kernel-hack-drill/drill_uaf_w_msg_msg.c

During the attack:



en.wikipedia.org/wiki/Rube_Goldberg_machine

1. Plug the holes in the initial and final slab caches
2. Allocate `(objs_per_slab * cpu_partial_slabs)` objects (for the per-CPU partial list)
3. Create a new active slab: allocate `objs_per_slab` objects
4. Allocate a vulnerable object
5. Create a new active slab again: allocate `objs_per_slab` objects
6. Free the slab with the UAF object: free `(objs_per_slab * 2 - 1)` objects before the last one
7. Fill up the partial list: free one of each `objs_per_slab` objects in the reserved slabs
8. Reclaim the UAF memory as a final slab (spray new objects) and exploit UAF

Debugging Cross-Cache Attack: Kernel Patch

```
diff --git a/ipc/msgutil.c b/ipc/msgutil.c
@@ -64,6 +64,7 @@ static struct msg_msg *alloc_msg(size_t len)
        msg = kmem_buckets_alloc(msg_buckets, sizeof(*msg) + alen, GFP_KERNEL);
        if (msg == NULL)
                return NULL;
+       printk("msg_msg 0x%lx\n", (unsigned long)msg);

        msg->next = NULL;
        msg->security = NULL;

diff --git a/mm/slub.c b/mm/slub.c
@@ -3140,6 +3140,7 @@ static void __put_partials(struct kmem_cache *s, struct slab *partial_slab)
        while (slab_to_discard) {
                slab = slab_to_discard;
                slab_to_discard = slab_to_discard->next;
+               printk("__put_partials: cache 0x%lx slab 0x%lx\n", (unsigned long)s, (unsigned long)slab);

                stat(s, DEACTIVATE_EMPTY);
                discard_slab(s, slab);
```

- `__put_partials()` calls `discard_slab()`, which moves the slab to the page allocator

Kernel Hack Drill: Cross-Cache Attack

Debugging Cross-Cache Attack: Console Output and GDB

- Legend: kernel log, stdout, GDB session (with bata24/gef)

```
[   49.755740] drill: kmalloc'ed item 5081 (0xffff8880068878a0, size 95)

[+] current_n: 5082 (next for allocating)
4) obtain dangling reference from use-after-free bug
[+] uaf_n: 5081

gef> slab-contains 0xffff8880068878a0
[+] Wait for memory scan
slab: 0xffffea00001a21c0
kmem_cache: 0xffff88800384e800
base: 0xffff888006887000
name: kmalloc-rnd-14-96  size: 0x60  num_pages: 0x1

[   51.371255] __put_partials: cache 0xffff88800384e800 slab 0xffffea00001a21c0
[   51.463570] msg_msg 0xffff8880068878a0
```

- The `drill_item_t` object `0xffff8880068878a0` in slab `0xffffea00001a21c0` is reallocated as `msg_msg`

Conclusion

Conclusion

1. **Linux Kernel Defence Map**  helps to:
   - Get an **overview** of Linux kernel security
   - Develop a **threat model** for your GNU/Linux system
   - Learn about kernel defences that **can help** against these threats

https://www.vecteezy.com/free-vector/cat

Conclusion

1. **Linux Kernel Defence Map**  helps to:
   - Get an **overview** of Linux kernel security
   - Develop a **threat model** for your GNU/Linux system
   - Learn about kernel defences that **can help** against these threats
2. **kernel-hardening-checker** ✅ helps to control the security-related parameters of your kernel

https://www.vecteezy.com/free-vector/cat

Conclusion

1. **Linux Kernel Defence Map**  helps to:
   - Get an **overview** of Linux kernel security
   - Develop a **threat model** for your GNU/Linux system
   - Learn about kernel defences that **can help** against these threats
2. **kernel-hardening-checker** ✅ helps to control the security-related parameters of your kernel
3. **kernel-hack-drill** 🎮 allows to experiment with Linux kernel vulnerabilities and security hardening features

https://www.vecteezy.com/free-vector/cat

Thanks / Спасибо!

Enjoy the conference!

Contacts:

a13xp0p0v

alex.popov@linux.com

Blog: a13xp0p0v.github.io

Channel: t.me/linkersec