Dashing Kernel Exploitation

Eduardo Vela & Jordy Zomer

whoamwe?

Eduardo Vela

Jordy Zomer

Artem Metla

Problem Statement 📢

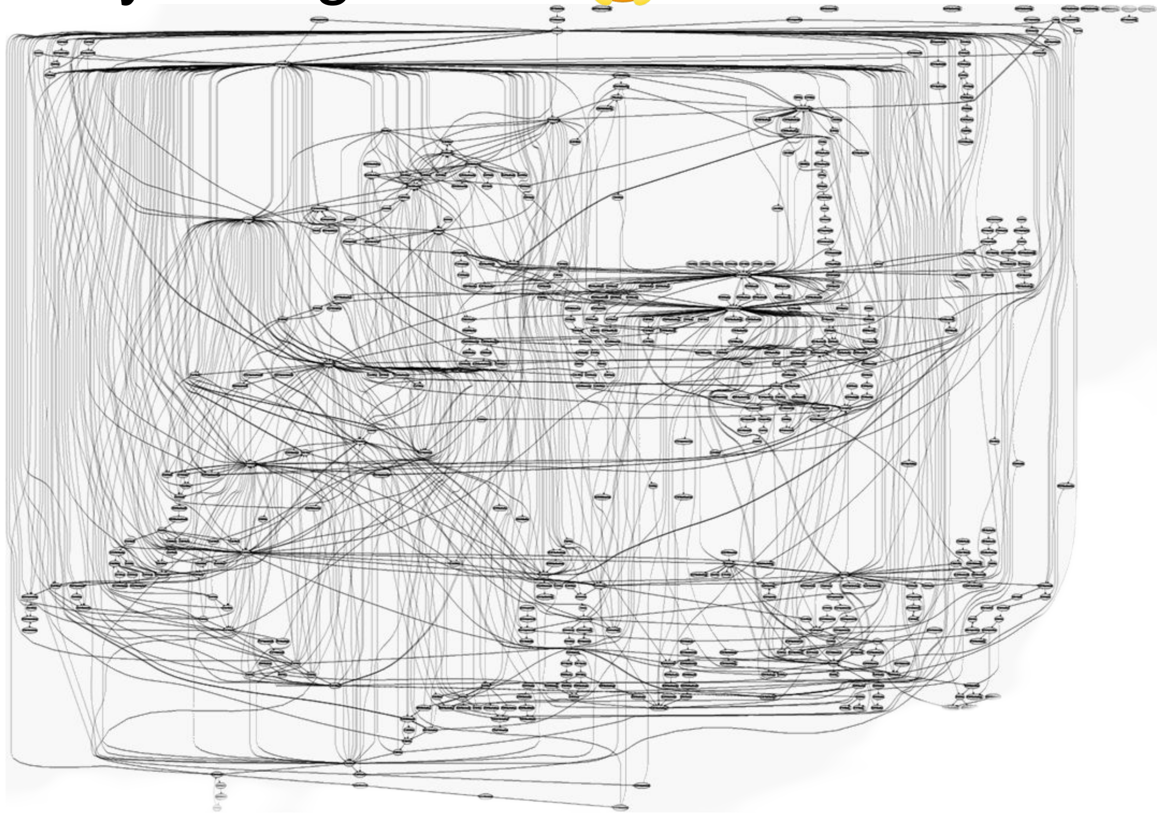
Analyzing the kernel is a real pain! It's like navigating a maze of tangled code and memory allocations. Just figuring out what triggers a bug or how an allocation of a certain object is done takes forever, even with the tools we have. They're helpful, but they only do part of the job, so most of the work still falls on us.

Problems: CVE overload 🤯

- Changes in Linux CNA make almost every bug a CVE
- Analysts are being 🌊 (flooded) with them!
- Makes it hard to prioritize and identify the most important ones

Problems: The Kernel Analysis Nightmare 😱

- Complex call-graphs
- Function pointers and indirect calls make it difficult to understand code flow
- Takes lots of time to figure out which restrictions a certain path has (syscall, capabilities,kernel configs, sysctl etc).



Problems: Heap Exploitation💥

"I have an out-of-bounds write at offset X of size Y. Which fields in the same kmalloc cache can I overwrite?"

"I have overwritten field X in struct Y, which is a function pointer. Where does it get called?"

"Which fields in this struct can be leaked to user space to gain an info-leak?"

"What are the allocation sizes, types, and flags used for related objects?"

“How do I trigger an allocation of struct foo?"

✨ Introducing the Kernel Exploitation Dashboard ✨

Reachability Dashboard: Key Features 🔑

- **Simplifies code navigation and function pointer analysis:** Trace function pointer usage and potential targets.
- **Identifies syscall triggers:** See which syscalls can lead to specific function execution.
- **Highlights syzkaller program interactions:** Pinpoint syzkaller programs that trigger lines within a function.
- **All-in-all:** Traverse the call-graph like a boss!

Structs Dashboard: Key Features 🔑

- **Reveals memory allocation details:** Understand which calls allocate specific structures, their sizes, and allocation flags.
- **Provides structure field insights:** Explore structure fields and their access types (read/write/execute).
- **Tracks field access locations:** Identify where structure fields are accessed throughout the code.

Tech Stack: Syzkaller 🛠️

Generates random system call sequences to test the kernel and uncover vulnerabilities.

Coverage Map: Maintains a "Coverage Map" to track which input reaches which code, used for prioritization.

Dashboard Integration:

- Extracts the coverage map from "syzbot" (a continuous Syzkaller instance reporting bugs).
- Uses the map to determine which program can reach a specific line of code.

Use-case: Helps identify how to trigger specific code paths in the kernel for analysis or exploitation.

Tech Stack: BTF (BPF Type Format) 🛠️

BTF is a debugging format that provides type information about kernel data structures.

BTF: embeds type information within the kernel image itself or as a separate file, allowing tools to understand the layout and types of kernel structures.

Dashboard Integration:

- The dashboard uses BTF to extract information about fields and structs, including their types, sizes, and offsets.

Use-case: This information is crucial for understanding heap objects, their relationships, and potential primitives you can achieve with them.

Tech Stack: BTF (BPF Type Format) 🛠️

```
+----------------------------------------------------------------------------------------------+
|{'id': 77, 'kind': 'STRUCT', 'name': 'list_head', 'size': 16, 'vlen': 2,                        |
|    'members': [                                                                                |
|        {'name': 'next', 'type_id': 78, 'bits_offset': 0},<--------------------------------+---
|        {'name': 'prev', 'type_id': 78, 'bits_offset': 64}                                  |   |
|    ]                                                                                       |   |
|}                                                                                           |   |
+----------------------------------------------------------------------------------------------+ |
                                                                                                 |
                                           ...                                                   |
                                                                                                 |
+----------------------------------------------------------------------------------------------+ |
|                                          ...                                                | |
+----------------------------------------------------------------------------------------------+ |
                                                                                                 |
                                           ...                                                   |
                                                                                                 |
+----------------------------------------------------------------------------------------------+ |
|{'id': 73060, 'kind': 'STRUCT', 'name': 'msg_msg', 'size': 48, 'vlen': 5,                    | |
|    'members': [                                                                            | |
|        {'name': 'm_list', 'type_id': 77, 'bits_offset': 0}, ------------------------------+--
|        {'name': 'm_type', 'type_id': 40, 'bits_offset': 128},                               |
|        {'name': 'm_ts', 'type_id': 64, 'bits_offset': 192},                                 |
|        {'name': 'next', 'type_id': 73062, 'bits_offset': 256},                              |
|        {'name': 'security', 'type_id': 88, 'bits_offset': 320}                              |
|    ]                                                                                        |
|}                                                                                            |
+----------------------------------------------------------------------------------------------+
```

```
+----------------------------------------------------+
| /* SPDX-License-Identifier: GPL-2.0 */             |
| #ifndef _LINUX_MSG_H                               |
| #define _LINUX_MSG_H                               |
|                                                    |
| #include <linux/list.h>                            |
| #include <uapi/linux/msg.h>                        |
|                                                    |
| /* one msg_msg structure for each message */       |
| struct msg_msg {                                   |
|     struct list_head m_list;                       |
|     long m_type;                                   |
|     size_t m_ts;        /* message text size */    |
|     struct msg_msgseg *next;                       |
|     void *security;                                |
|     /* the actual message follows immediately */   |
|}                                                   |
+----------------------------------------------------+
```

Tech Stack: BTF (BPF Type Format) 🛠️

```
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|  Struct Name  |  Struct Size  |  Parent Type  |  Kind  |           Type           |        Name        |  Bits Offset  |  Bits Nr  |  Bits End  |  Elastic?  |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| msg_msg       | 48            | list_head     | PTR    | struct list_head *       | m_list.next        | 0             | 64        | 64         | No *       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| msg_msg       | 48            | list_head     | PTR    | struct list_head *       | m_list.prev        | 64            | 64        | 128        | No *       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| msg_msg       | 48            | msg_msg       | INT    | long int                 | m_type             | 128           | 64        | 192        | No *       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| msg_msg       | 48            | msg_msg       | INT    | long unsigned int        | m_ts               | 192           | 64        | 256        | No *       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| msg_msg       | 48            | msg_msg       | PTR    | struct msg_msgseg *      | next               | 256           | 64        | 320        | No *       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| msg_msg       | 48            | msg_msg       | PTR    | void *                   | security           | 320           | 64        | 384        | No *       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

* We can’t determine that object is elastic based on BTF data only in this case. Static analysis should be used to check the way the objects are allocated.

Tech Stack: CodeQL 🛠️

CodeQL is a static analysis engine that allows you to query codebases to find vulnerabilities and better understand code.

How it works:

- CodeQL treats *code* as **data**. This means you can write queries to explore the codebase and find potential issues.
- It builds a database representation of the code, enabling all sorts of analysis:
    - ControlFlow (In which order is code executed, such as “what is executed next”)
    - DataFlow (Finds a flow from source data to a sink where the data isn’t modified)
    - TaintTracking (Finds a flow from source data that can be modified to a sink)
    - etc.
- Does semantic analysis (has context about the code, instead of simple pattern matching)

🏊Deep-Dive: CallGraph ☎️

ControlFlowNode: The Building Blocks of the CallGraph

- What is a ControlFlowNode?

ControlFlowNode: The Building Blocks of the CallGraph

Example: Code

```
x = 5

if x > 10:

  print("x is greater than 10")

else:

  print("x is not greater than 10")
```

Example: Control Flow Graph

```
[Start] --> [x = 5] --> [if x > 10]

                              |

                  (yes) -------'

                  |

                  V

              [print("x is greater than 10")] --> [End]

                  ^

                  |

                  (no) --------

                              |

[print("x is not greater than 10")] --> [End]
```

CallGraph: Identifying Calls within Functions ☎️

Query:

```
query predicate edges(ControlFlowNode a, ControlFlowNode b) {

    a = b.(Call).getEnclosingFunction()

}
```

Execution:

```
+--------------------------------------------------------------+      --------------------+
|                                                              |      |         +---------+------------------+
|                                                              |      |         |(Call)getEnlosingFunction()|
|                                                              |      |          +--------------------------+
|   SYSCALL_DEFINE0(getuid)                                    |  <----                    ^
|   {                                                          |                           |
|    /* Only we change this so SMP safe */                     |               +-----------+-----------+
|    return from_kuid_munged(current_user_ns(), current_uid());+---------------| from_kuid_munged(Call)|
|   }                                                          |               +-----------------------+
|                                                              |
|                                                              |
+--------------------------------------------------------------+
```

CallGraph: Where Does the Call Go? 📞

- Introducing `resolveCall`
    - Direct Calls
    - Function Pointers (simple-dataflow)
    - Virtual Calls

Query:

```
query predicate edges(ControlFlowNode a, ControlFlowNode b) {
    a = b.(Call).getEnclosingFunction() or
    b = resolveCall(a)
  }
```

CallGraph: Where Does the Call Go? 📞

What it looks like!

```
+---------------------------------------------------------------+      ----------------------+
|                                                               |      |
|                                                               |      |        +----------+------------------+
|                                                               |      |        |(Call)getEnlosingFunction()|
|   SYSCALL_DEFINE0(getuid)                                     |  <----        +---------------------------+
|   {                                                            |                        ^
|    /* Only we change this so SMP safe */                       |                        |
|    return from_kuid_munged(current_user_ns(), current_uid());+--------------|from_kuid_munged(Call) |
|   }                                                            |               +------------+----------+
|                                                                |                            |
|                                                                |                            |
+---------------------------------------------------------------+                             |
                                                                                               v
                                                                                      +-----------+
                                                                                      |resolveCall|
                                                                                      +-----+-----+
+--------------------------------------------------------------------------------------+   |
|   static inline uid_t from_kuid_munged(struct user_namespace *to, kuid_t kuid) | <--------+
|   {                                                                                    |
|    uid_t uid = from_kuid(to, kuid);                                                   |
|    if (uid == (uid_t)-1)                                                              |
|     uid = overflowuid;                                                                |
|    return uid;                                                                        |
|   }                                                                                    |
+--------------------------------------------------------------------------------------+
```

Graph:

```
+-----------------+
|function getuid()|
+--------+--------+
         v
+--------+-----------------+
|call to from_kuid_munged()|
+--------+-----------------+
         v
+---------------------------+
|function from_kuid_munged()|
+---------------------------+
```

🚨Problems!🚨

CallGraph: Where Does the FunctionPointer Go? 📞

- PointsToAnalysis
- Steengaard’s Algorithm
- Still false-positives 🥺

Query:

```
cached predicate exprCallEdge(ExprCall a, Function b) {
  a.getExpr().(TargetPointsToExpr).pointsTo() = b
}
```

Result:

```
+-----------------------------------------------------------------------------+
|static inline unsigned int io_apic_read(unsigned int apic, unsigned int reg) |
|{                                                                            |
| return x86_apic_ops.io_apic_read(apic, reg);-----+                          |
|}                                                 |                          |
+--------------------------------------------------+--------------------------+
                                                   v
                         +-------------------------------+
              +----------+TargetPointsToExpr(io_api_read)|
              |          +-------------------------------+
              v
+----------------------------------------------------------------------+
|unsigned int native_io_apic_read(unsigned int apic, unsigned int reg) |
|{                                                                     |
| struct io_apic __iomem *io_apic = io_apic_base(apic);                |
| writel(reg, &io_apic->index);                                        |
| return readl(&io_apic->data);                                        |
|}                                                                     |
|                                                                      |
+----------------------------------------------------------------------+
```

CallGraph: Taming the Function Pointers 🦁

- There’s still false-positives, how do we fix this??

CallGraph: Taming the Function Pointers 🦁

- There's still false-positives, how do we fix this??
- We have additional context 😎👌🔥

```
cached predicate exprCallEdge(ExprCall a, Function b) {
  a.getExpr().(TargetPointsToExpr).pointsTo() = b and
  a.getExpr().(TargetPointsToExpr).confidence() >= 0.2 and
  // Get the number of parameters of the function
  exists(int numParams |
    numParams = count(b.getParameter(_)) and
    // Iterate over each parameter
    forall(int i | i in [0 .. numParams - 1] |
      exists(Parameter p | p = b.getParameter(i) |
        // Get the type of the parameter
        exists(Type paramType | paramType = p.getType() |
          // Get the argument at the corresponding index
          exists(Expr arg | arg = a.getArgument(i) |
            // Check if the argument's type is compatible
            arg.getType().(PointerType).getBaseType() = paramType
            or
            arg.getType() = paramType
          )
        )
      )
    )
  )
}
```

CallGraph: Filling The Gaps!

- Some function pointers are missing
- Mostly class aggregate literals

```
static const struct file_operations kprobe_events_ops =
{
    .owner          = THIS_MODULE,
    .open           = probes_open,
    .read           = seq_read,
    .llseek         = seq_lseek,
    .release        = seq_release,
    .write        = probes_write,
};
```

```
import cpp

class OpsAggregateLiteral extends ClassAggregateLiteral {
  Field field;

  OpsAggregateLiteral() {
    exists(this.getAFieldExpr(field)) and field.getType() instanceof FunctionPointerIshType
  }

  Field getField() { result = field }
}

from OpsAggregateLiteral opal, Field field, Function fun, ExprCall call
where
  field = opal.getField() and
  fun = opal.getAFieldExpr(field).(FunctionAccess).getTarget() and
  call.getExpr() = field.getAnAccess()
select opal, field, fun, call
```

🏊Deep-Dive: Allocations 📚

Allocations: Finding Calls! 📲

- The most common allocation functions are `kmalloc`, `kzalloc` and `kvmalloc`.
- These functions take a size argument to determine how big the allocation should be.
- And a flags argument, the most important ones are GFP_KERNEL and GFP_ACCOUNT, which decide in what kind of slab cache they are allocated.

Example:

```
struct foo *ptr;

ptr = kmalloc(sizeof(struct foo), GFP_KERNEL);
if (!ptr)
        /* handle error ... */
```

Flags:

GFP_KERNEL -> kmalloc-* caches
GFP_KERNEL_ACCOUNT -> kmalloc-cg-* caches

Allocations: Finding Sizes! 💺🪑💺

- The size of the allocation decides which cache they end up in
- Known if it’s not dynamically sized

Example:

```
class KmallocCall extends FunctionCall {
  KmallocCall() {
    this.getTarget().hasName(["kmalloc", "kzalloc",
"kvmalloc"])
  }

  Expr getSizeArg() { result = this.getArgument(0) }

  string getSize() {
    if this.getSizeArg().isConstant()
    then result = this.getSizeArg().getValue()
    else result = "unknown"
  }
}
```

Allocations: Finding Flags! 🚩

- The flags also influence which cache is allocated from
  - GFP_KERNEL -> kmalloc-*
  - GFP_ACCOUNT -> kmalloc-cg-*

Example:

```
class KmallocCall extends FunctionCall {
  KmallocCall() {
    this.getTarget().hasName(["kmalloc", "kzalloc", "kvmalloc"])
  }
  Expr getSizeArg() { result = this.getArgument(0) }
  string getSize() { /* previous implementation */}

  string getFlag() {
    result =
      concat(Expr flag |
        flag = this.getArgument(1).getAChild*() and
        flag.getValueText().matches("%GFP%")
           |
        flag.getValueText(), "|"
      )
  }
}
```

Allocations: Finding Types! 💬

- sizeof() expressions
- Return types

Example:

```
class KmallocCall extends FunctionCall {
 /* other predicates removed to preserve screen space */
  Type sizeofParam(Expr e) {
    result = e.(SizeofExprOperator).getExprOperand().getFullyConverted().getType()
    or
    result = e.(SizeofTypeOperator).getTypeOperand()
  }

  Struct getStruct() {
    exists(Expr sof |
      this.getSizeArg().getAChild*() = sof and
      this.sizeofParam(sof) = result
    ) or
   result = this.getFullyConverted().getType().stripType()
  }
}
```

🏊Deep-Dive: Fields🌾

Fields: Read/Write/eXecute 📖✍️⚡

- Execute (Function pointer calls of fields)
- Reads of fields
- Writes to field (modifications)

```
class FieldExec extends InterestingFieldAccesses {
    FieldExec() { exists(ExprCall ec | ec.getExpr() = this)}
    override string accessType() {result = "exec"}
}


class FieldWrite extends InterestingFieldAccesses {
    FieldWrite() { this.isModified()}
    override string accessType() {result = "write"}
}

class FieldRead extends InterestingFieldAccesses {
    FieldRead() { this.isRValue() }
    override string accessType() {result = "read"}
}
```

Fields: Info-Leaks🚰

- Field access flows to
  - Copy_to_user output
  - Put_user output
  - Printk output

```
class PrintkArgs extends LeakNode {
  PrintkArgs() {
    exists(FunctionCall fc |
      fc.getAnArgument() = this.asExpr() and fc.getTarget().hasName("printk")
    )
  }
}
class CopyToUserOut extends LeakNode {
  CopyToUserOut() {
    exists(FunctionCall fc |
      fc.getTarget().hasName("copy_to_user") and fc.getArgument(1) = this.asExpr()
    )
  }
}
class PutUser extends LeakNode {
  PutUser() {
    exists(FunctionCall fc |
      fc.getTarget().hasName("put_user") and fc.getArgument(0) = this.asExpr()
    )
  }
}
```

Fields: Info-Leaks🚰

- Dataflow
  - Souce (FieldAccess)
  - Sink (LeakNode)

```
module LeakFlowConfiguration implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) { source.asExpr() instanceof FieldAccess }

  predicate isSink(DataFlow::Node sink) { sink instanceof LeakNode }
}

module LeakFlow = TaintTracking::Global<LeakFlowConfiguration>;
```

```
                  +-------+
         +--------|Step 2 |
         |        +-------+
+--------+---------------------------------------------------------------------+
|ssize_t | rpc_pipe_generic_upcall(struct file *filp, struct rpc_pipe_msg *msg,| +---------+
|    char|__user *dst, size_t buflen)        +-------------------------------+-| | Step 1 |
|{       |                                   |                                 | +---------+
| char *data = (char *)msg->data + msg->copied;                                |
| size_t mlen = min(msg->len - msg->copied, buflen);                           |  +--------+
| unsigned long left;         +------------------------------------------------+--| Step 3|
|                             |                                                |  +--------+
| left = copy_to_user(dst, data, mlen);                                        |
| if (left == mlen) {                                                          |
|  msg->errno = -EFAULT;                                                       |
|  return -EFAULT;                                                             |
| }                                                                            |
|                                                                              |
| mlen -= left;                                                                |
| msg->copied += mlen;                                                         |
| msg->errno = 0;                                                              |
| return mlen;                                                                 |
|}                                                                             |
+------------------------------------------------------------------------------+
```

📅Future Plans📅

- Publicly available
- Open source
- Integration with other tooling
- Features:
  - DataFlow from Syscall to field assignments
  - DataFlow from Field to free
  - DataFlow from user to function pointers
- AI?

Code

- Somewhere in the coming weeks the code will appear on https://github.com/google/security-research/tree/master/analysis/kernel
- We will also host a public version that you can explore

💡 Questions? 💡