

Preview

# Linux

## Core Dump Analysis

### Advanced

**Data Structures and Generative AI**

Dmitry Vostokov  
Software Diagnostics Services

# Course Idea

- Missing topics in Accelerated Linux Core Dump Analysis
- Advanced Windows Memory Dump Analysis
- Extended Windows Memory Dump Analysis
- My interest in data science and visualization (trace and log analysis)
- My interest in machine learning and AI

# Course Purpose

- Modern perspective (2026)
- Focus on concepts, not details
- Integration with coding assistants

# Coding Assistants Era

- ⦿ GDB extensions can be generated and tested from specs
- ⦿ Data science, visualization, AI, and ML postprocessing scripts can be generated as needed

# Training Goals

- Develop GDB extensions
- Learn how to navigate data structures
- Use data processing, analysis, and visualization
- Use Generative AI
- Use UML for communication
- Learn fundamentals of kernel modules

# Schedule

- ◉ Navigating lists
- ◉ Writing GDB extensions (Python)
- ◉ Raw region analysis
- ◉ Kernel modules and device drivers
- ◉ Data analysis and visualization
- ◉ Generative AI and MCP

# Training Principles

- ⦿ Talk only about what I can show
- ⦿ Lots of pictures
- ⦿ Lots of examples
- ⦿ Original content and examples

# Prerequisites

- Basic and intermediate-level Linux core dump analysis
- Basic C and Python knowledge
- Basic x64 and ARM64 disassembly knowledge

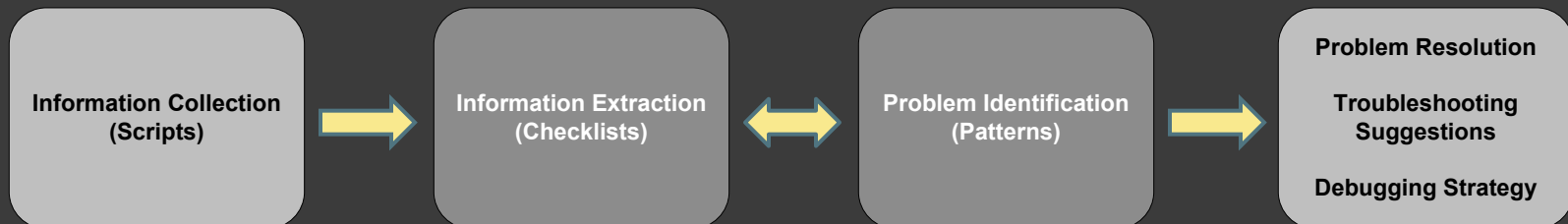
# Pattern-Oriented Diagnostic Analysis

**Diagnostic Problem:** a set of indicators (symptoms, signs) describing a problem.

**Diagnostic Pattern:** a common recurrent identifiable problem together with a set of recommendations and possible solutions to apply in a specific context.

**Diagnostic Analysis Pattern:** a common recurrent analysis technique and method of diagnostic pattern identification in a specific context.

**Diagnostics Pattern Language:** common names of diagnostic and diagnostic analysis patterns. The same language for any operating system: Windows, macOS, Linux, ...



# Practice Exercises

# Links

- Core Dumps:

Included in Exercise 0

- Exercise Transcripts:

Included in this book

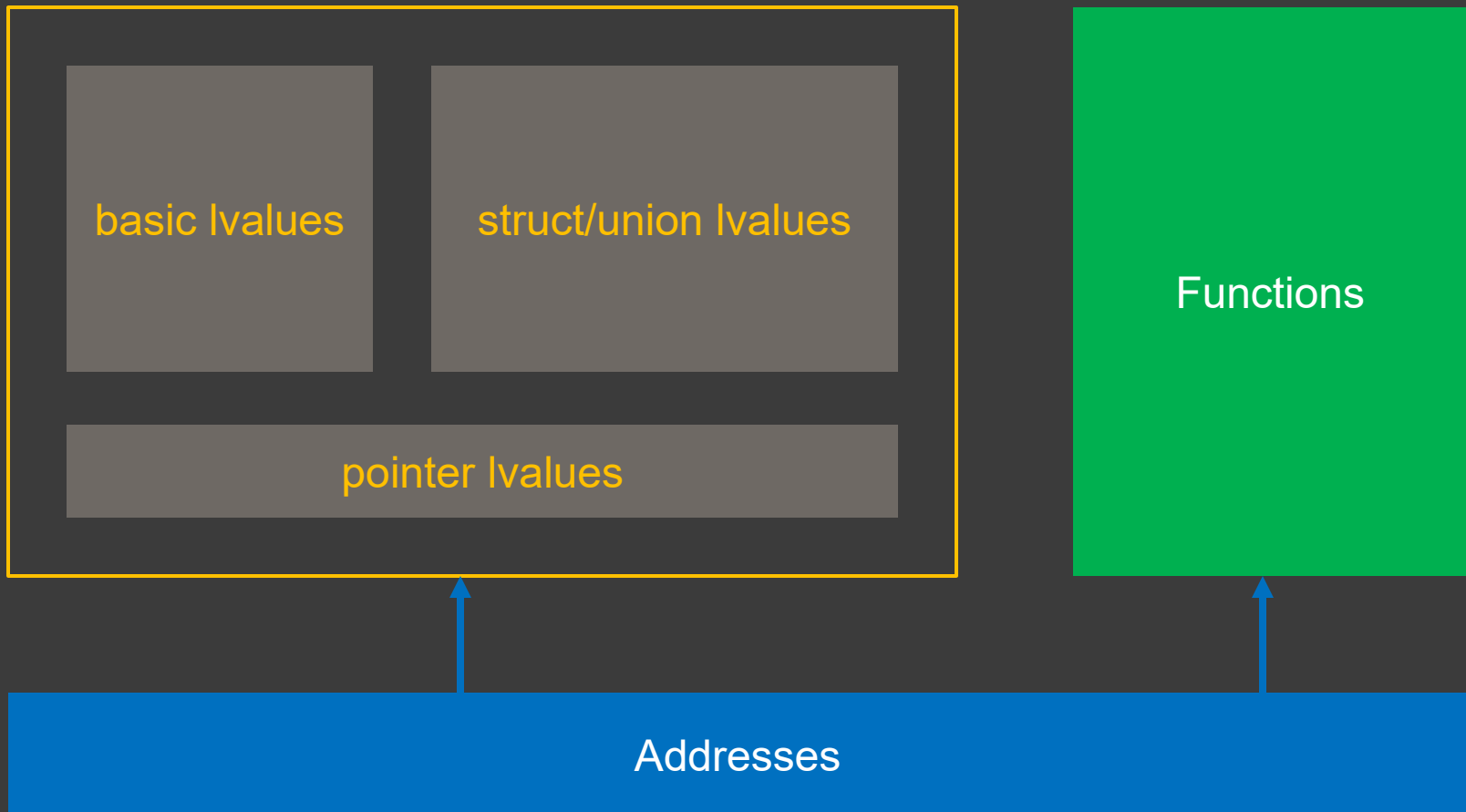
# Exercise 0

- ⦿ **Goal:** Install GDB and crash tool, and check if both load core dumps correctly
- ⦿ [\AdvLCDA-Dumps\Exercise-0.pdf](#)

# A Crash **Dump** Course in C Language

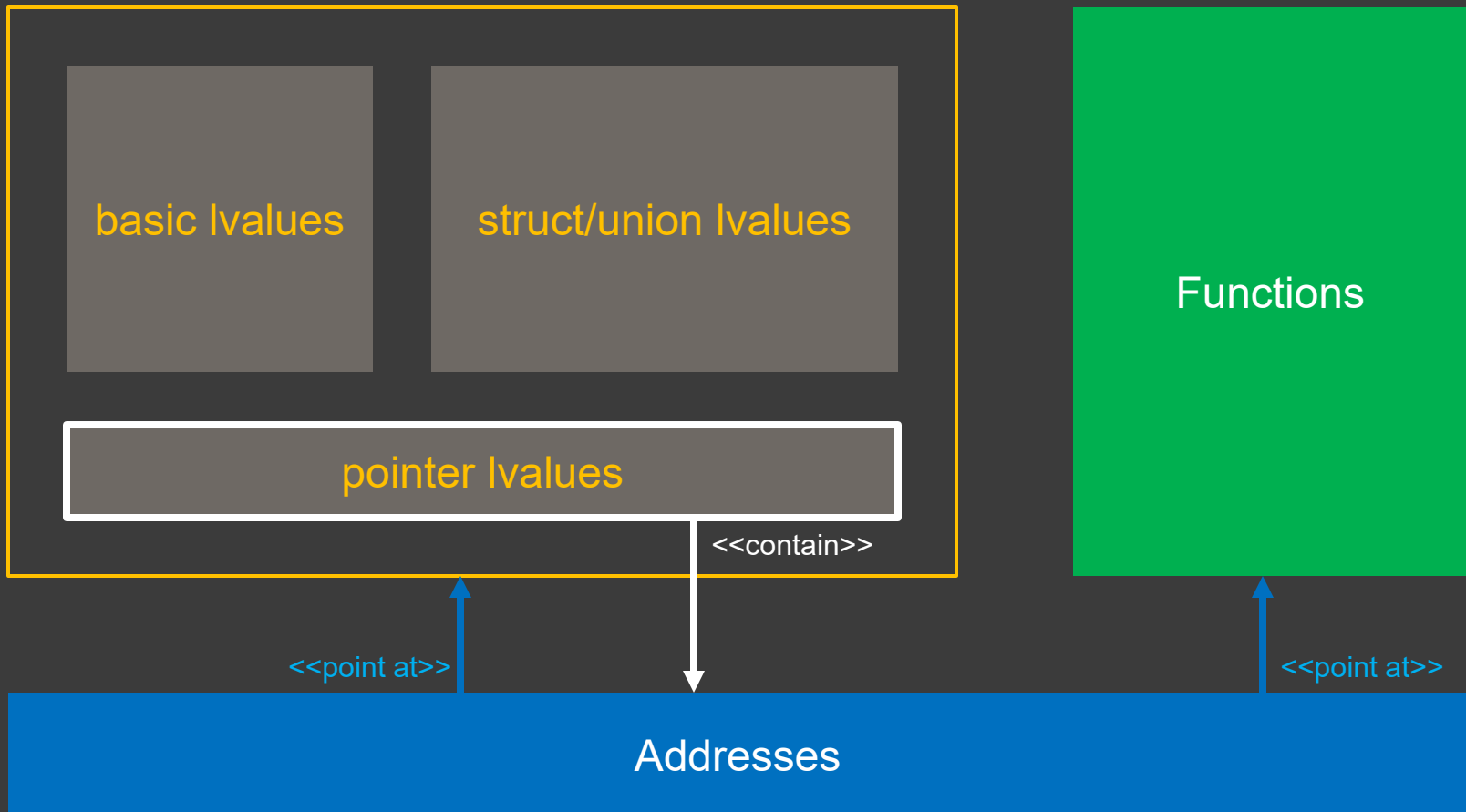
# lvalues, Functions, Addresses

```
int lvalue; lvalue = 1; // = rvalue  
int lvalue2 = lvalue;
```



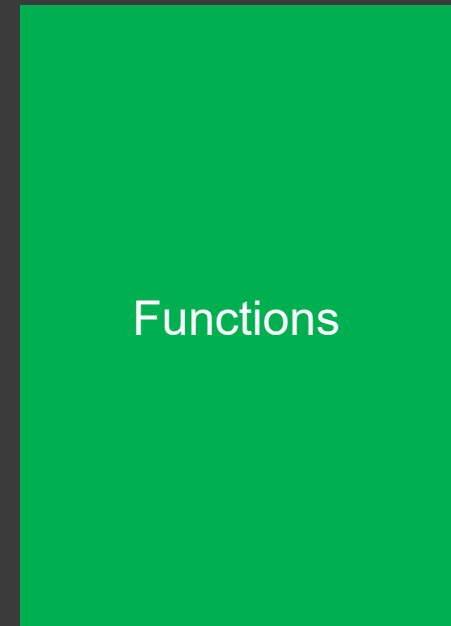
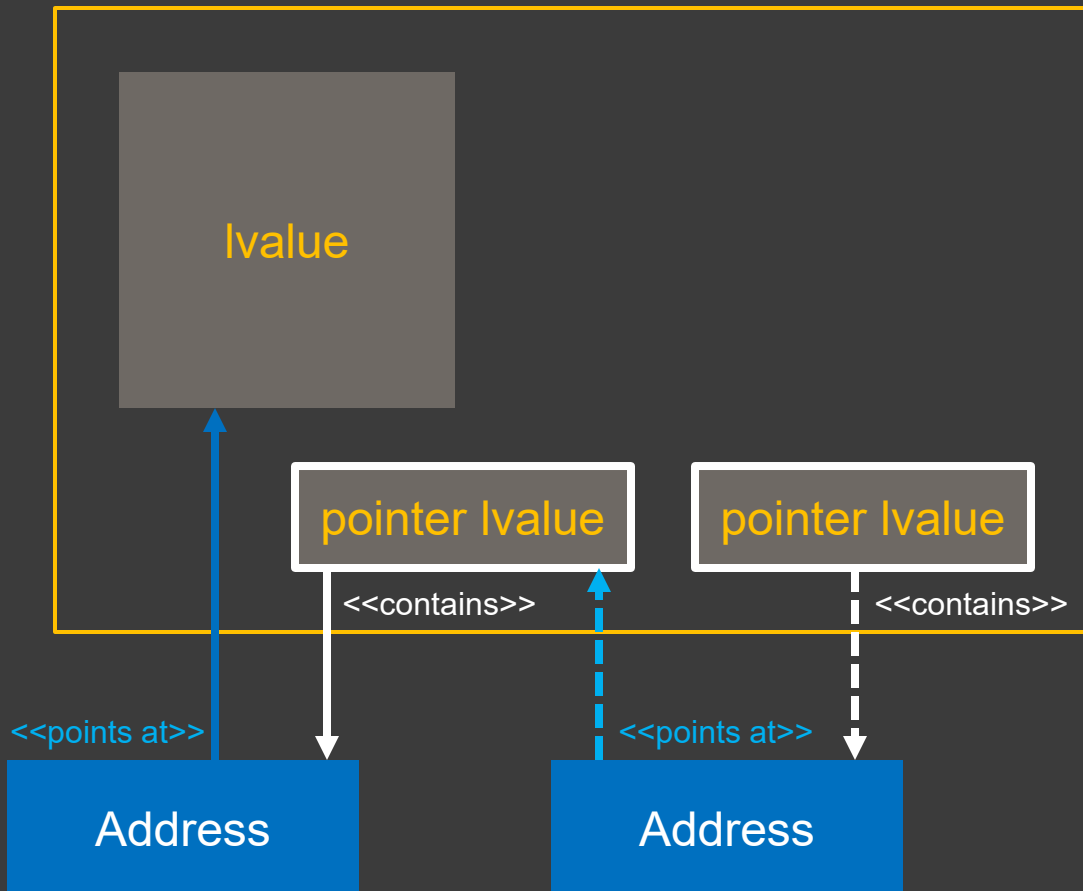
# Pointers and Addresses

```
int lvalue = 0;  
int* p_lvalue = &lvalue;
```



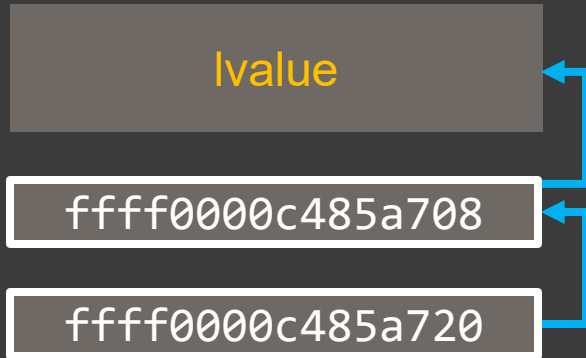
# Pointers to Pointers

```
int lvalue = 0; int* p_lvalue = &lvalue;  
int** pp_lvalue = &p_lvalue;
```



# Addresses and Memory Contents

```
...
ffff0000c485a700:
ffff0000c485a708:
ffff0000c485a710:
ffff0000c485a718:
ffff0000c485a720:
ffff0000c485a728:
ffff0000c485a730:
...
...
...
ffff8000815e7ed0:
ffff8000815e7ed8:
ffff8000815e7ee0:
ffff8000815e7ee8:
ffff8000815e7ef0:
ffff8000815e7ef8:
...
```



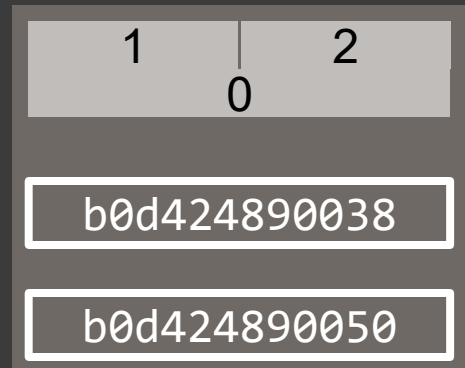
# Deductive Summary

- ⦿ lvalues (and functions) have memory addresses → lvalues and functions have memory content
- ⦿ Memory addresses are rvalues → memory addresses don't have their own memory addresses
- ⦿ Pointers' memory content contains addresses
- ⦿ Every number (rvalue) can be interpreted as a memory address → an lvalue that contains a number can be interpreted as a pointer
- ⦿ Pointers are lvalues → pointers have their own memory addresses
- ⦿ Pointers may contain addresses of other pointers that contain addresses of ...

# C Structs

```
struct lvalue {
    struct {
        int field;
        int field2;
        long field3;
    } substruct;
    long dummy;
    struct lvalue* p_field;
    long dummy3;
    struct lvalue** pp_field;
} lvalue = {
    { .field = 1,
      .field2 = 2,
      .field3 = 0
    },
    .p_field = &lvalue,
    .pp_field = &lvalue.p_field
};
```

```
...
b0d424890030:
b0d424890038:
b0d424890040:
b0d424890048:
b0d424890050:
b0d424890058:
b0d424890060:
...
```



# Summary for Next Sections

- ⦿ Every substructure or field has an offset
- ⦿ If we have a field address, to get the structure address, subtract the field offset
- ⦿ If we have a structure address, to get a field address, add the field offset
- ⦿ For C code, we have macros; for GDB, use the mental math above, commands, and scripts

# A Crash **Dump** Course in Unified Modeling Language

## Part I

# Classes and Objects (General)

<b>class</b>
+field_public : type_a
-field_private : type_b
+method_public()
-method_private()

<b><u>object a : class</u></b>
field_public = value1
field_private = value2

<b><u>object b : class</u></b>
field_public = value3
field_private = value4

# Data Types as Classes (General)

```
data_type  
+field_a : type_a  
+field_b : type_b
```

```
object a : data type  
field_a = value1  
Field_b = value2
```

```
object b : data type  
field_a = value3  
field_b = value4
```

# C Structs and Values

```
struct a  
+field : struct b  
+field2 : int
```

```
a1 : struct a  
field = value1  
Field2 = 1
```

```
a2 : struct a  
field = value2  
Field2 = 2
```

# Memory as Objects

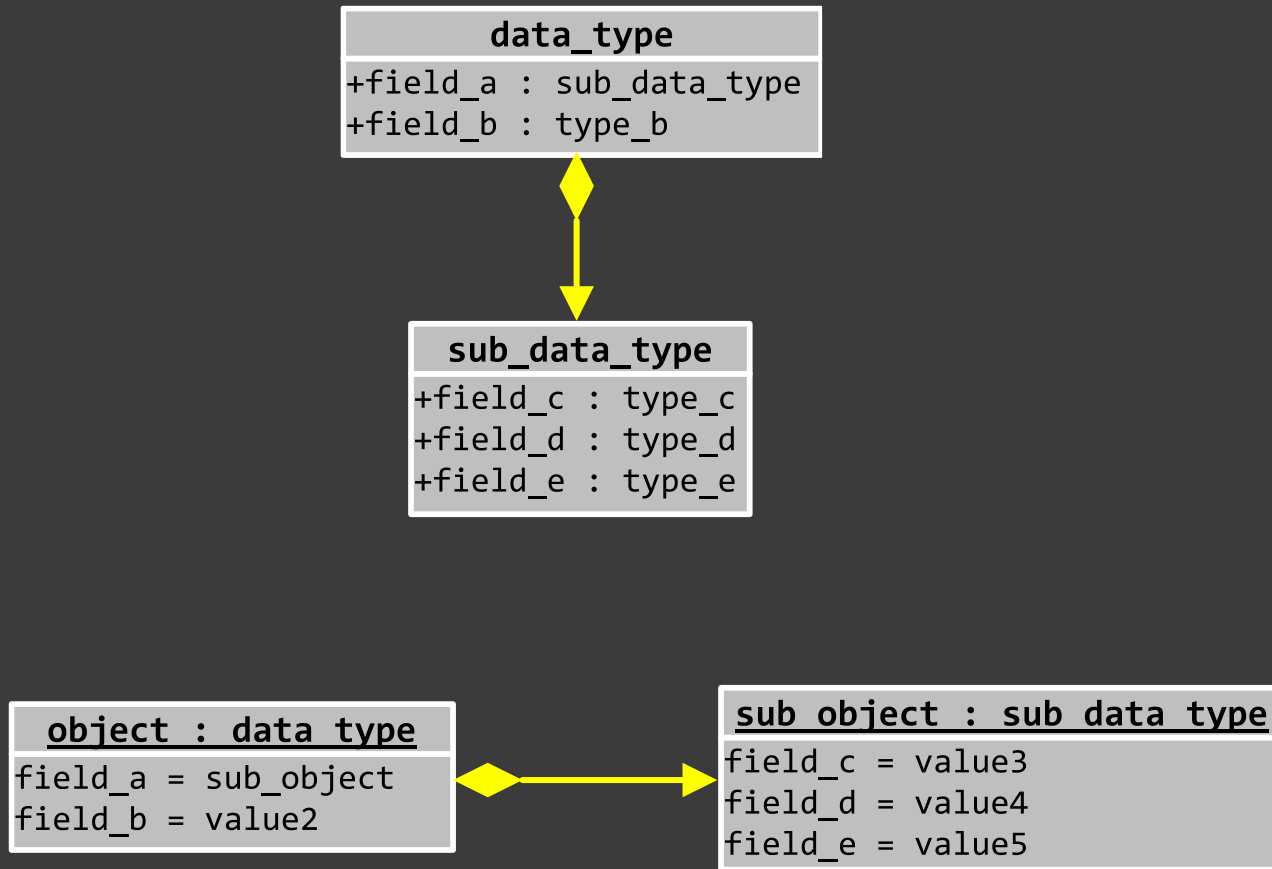
```
crash> struct task_struct
```

```
struct task_struct {  
    struct thread_info thread_info;  
    unsigned int __state;  
    ...
```

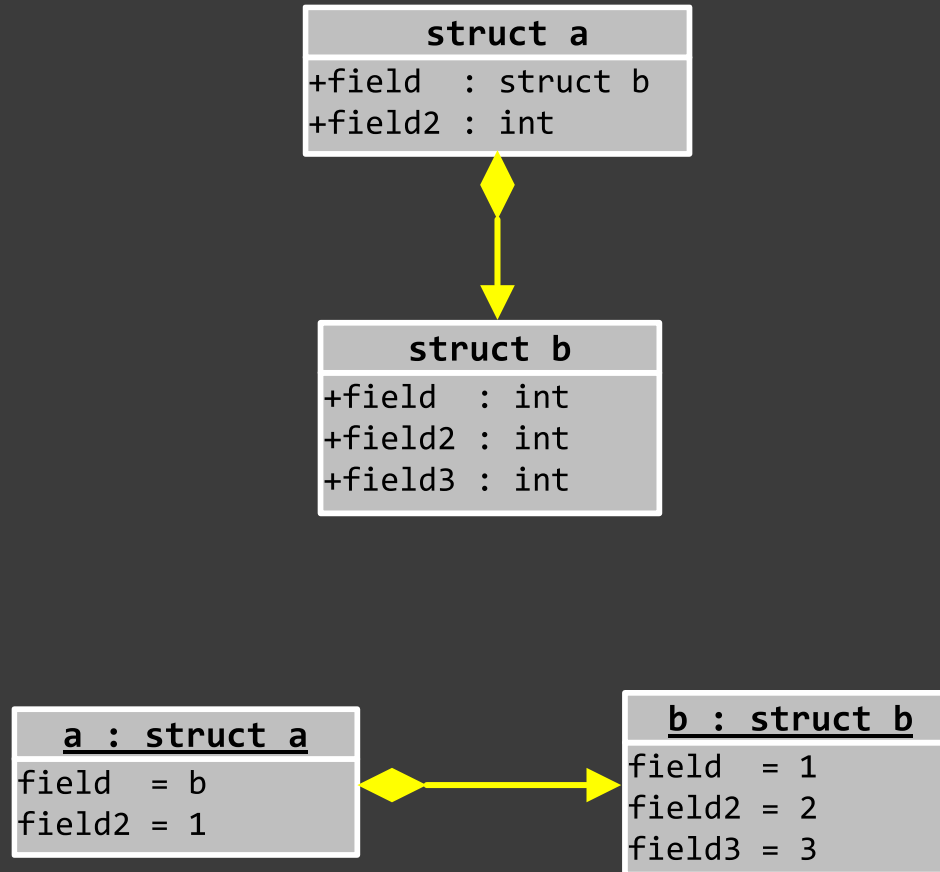
```
crash> struct task_struct ffff800083239ac0
```

```
struct task_struct {  
    thread_info = {  
        flags = 8,  
        ttbr0 = 13620928512,  
        {  
            preempt_count = 4294967296,  
            preempt = {  
                count = 0,  
                need_resched = 1  
            }  
        },  
        cpu = 0  
    },  
    __state = 0,  
    ...
```

# Aggregation (General)



# Aggregation (C Structs)

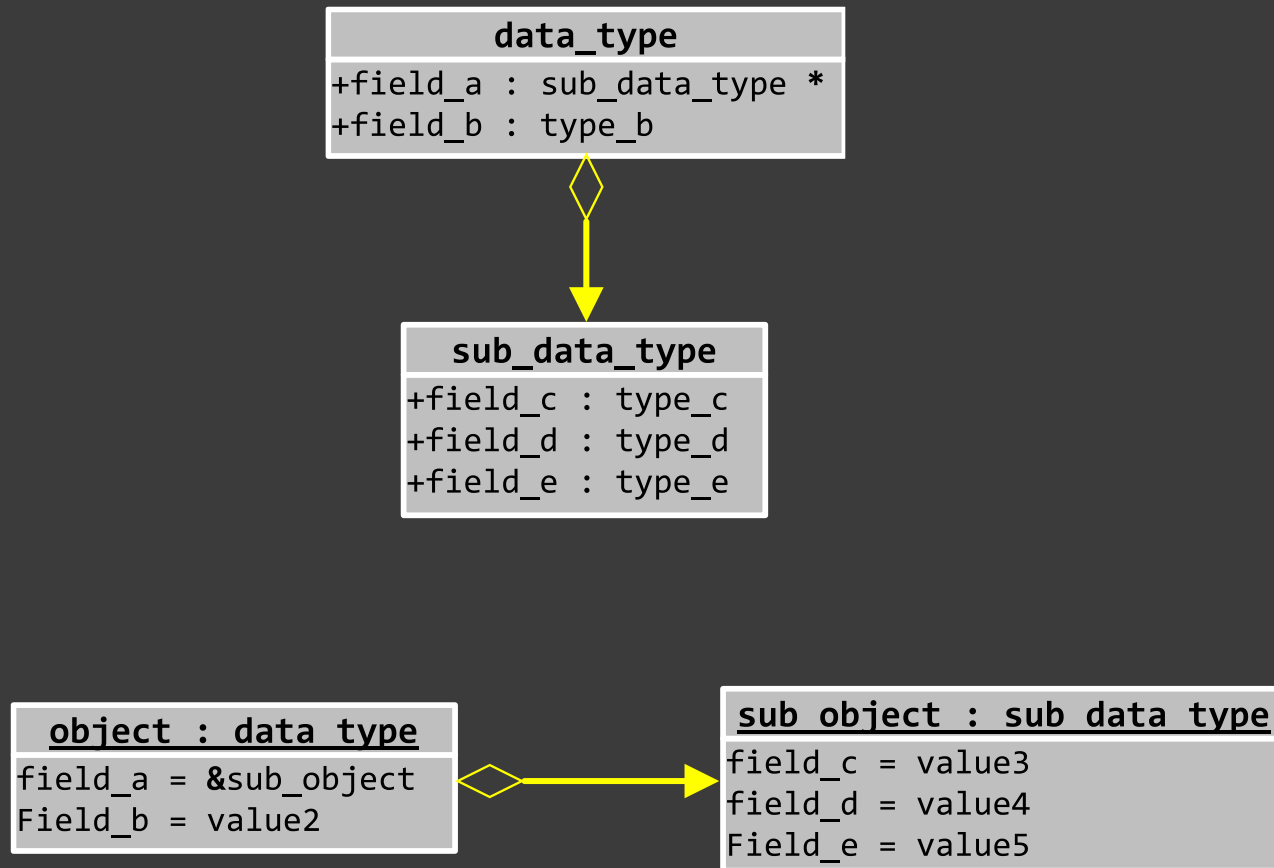


# Aggregation in Memory

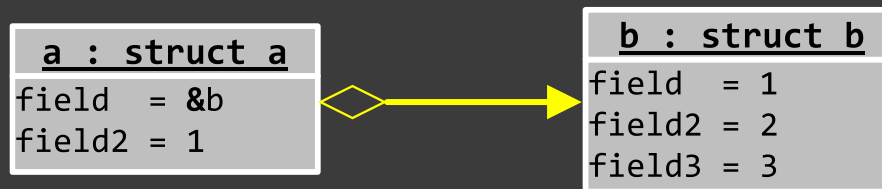
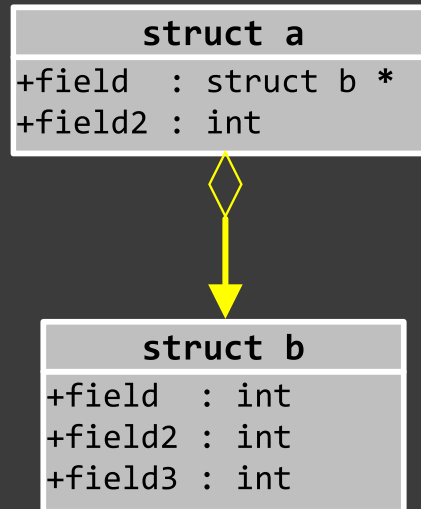
```
crash> struct task_struct -o
struct task_struct {
    [0] struct thread_info thread_info;
    [32] unsigned int __state;
    [36] unsigned int saved_state;
    ...
}
```

```
crash> struct thread_info -o
struct thread_info {
    [0] unsigned long flags;
    [8] u64 ttbr0;
    union {
    [16]     u64 preempt_count;
            struct {
                u32 count;
                u32 need_resched;
    [16]     } preempt;
    };
    [24] u32 cpu;
}
SIZE: 32
```

# Composition (General)



# Composition (C Structs)



# Composition in Memory

```
crash> struct task_struct
```

```
struct task_struct {
```

```
...
```

```
    struct task_struct *parent;
```

```
...
```

```
crash> struct task_struct ffff0000c0584400
```

```
...
```

```
parent = 0xffff0000c03ba200
```

```
...
```

```
crash> struct task_struct 0xffff0000c03ba200
```

```
struct task_struct {
```

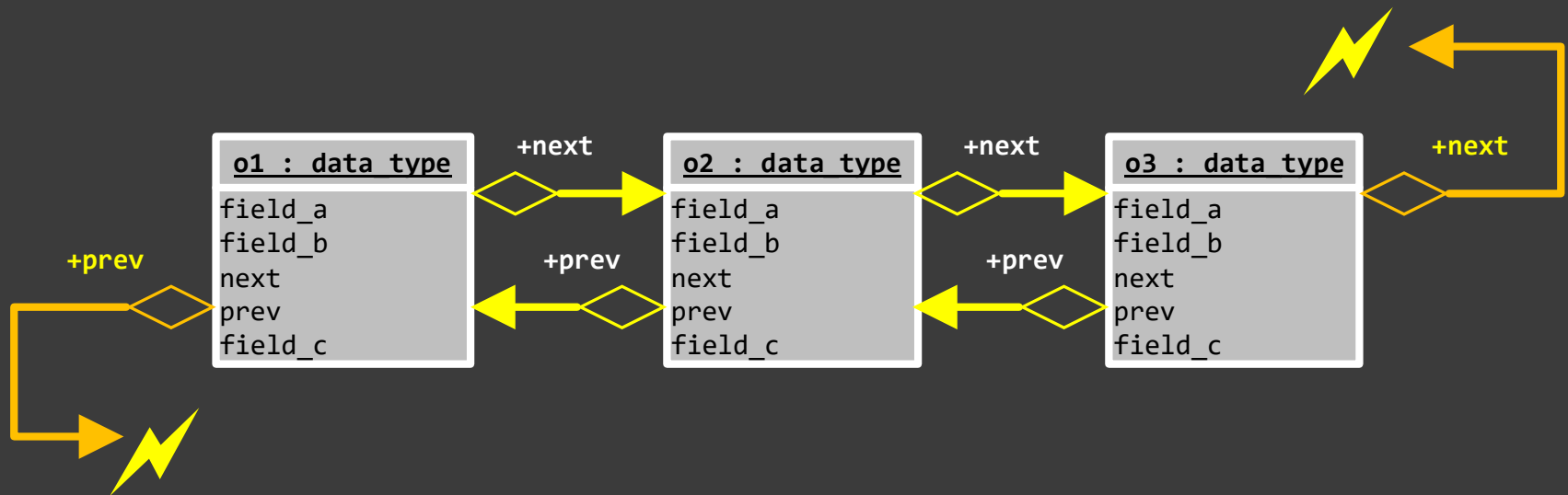
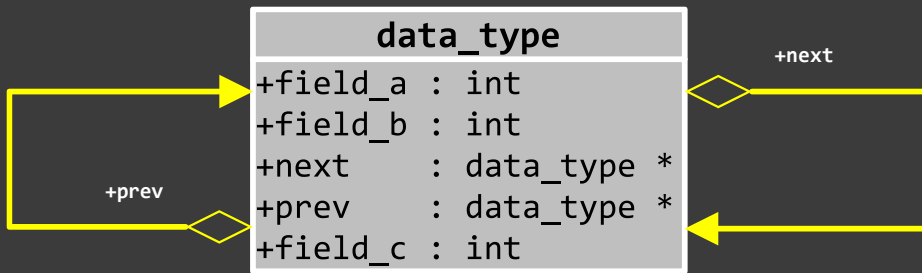
```
...
```

```
parent = 0xffff800083239ac0 <init_task>,
```

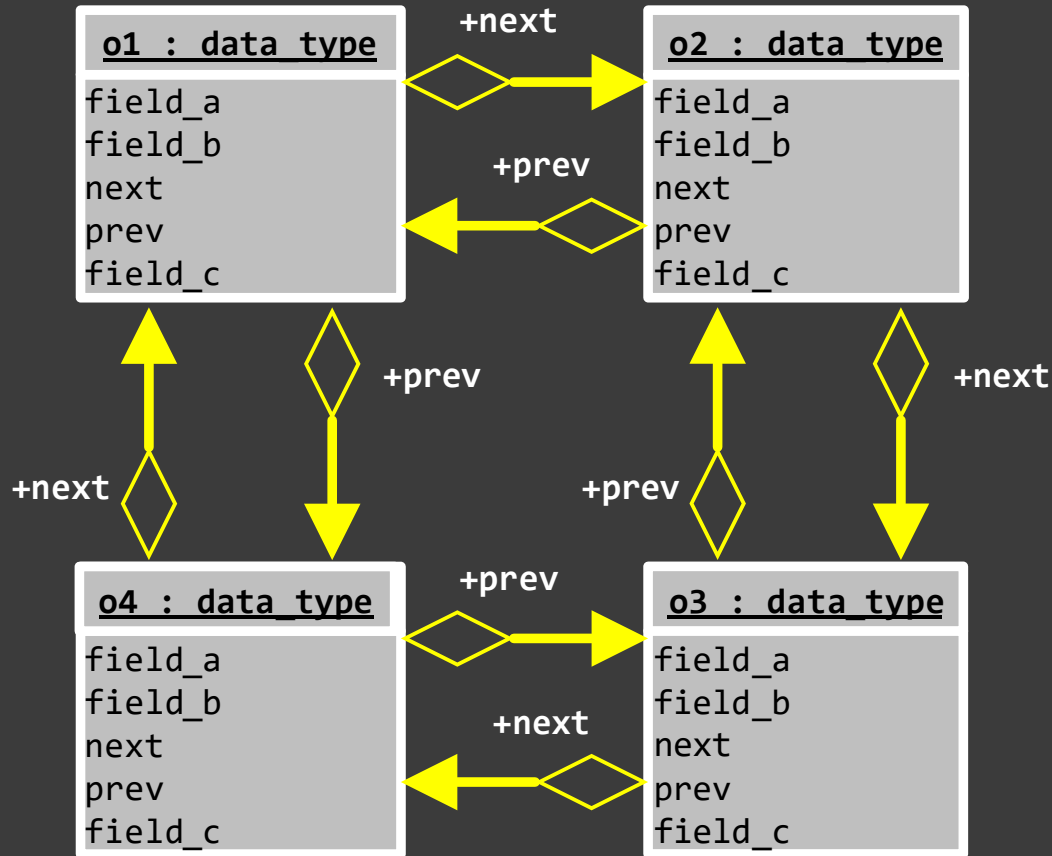
```
...
```

# Linked Lists

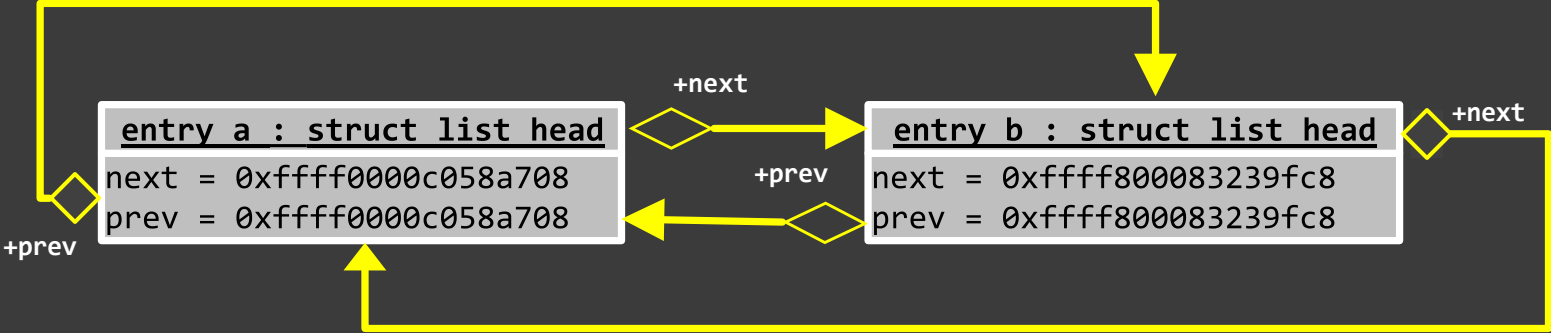
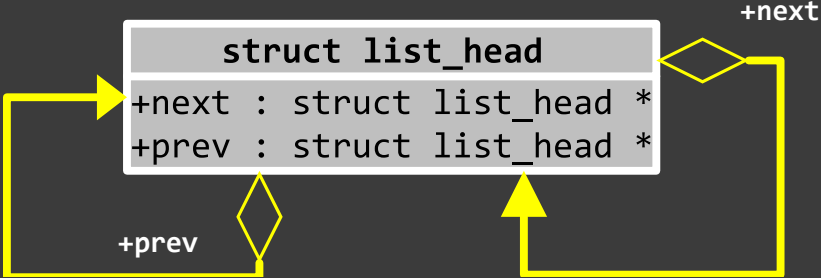
# Linked List



# Circular Linked List

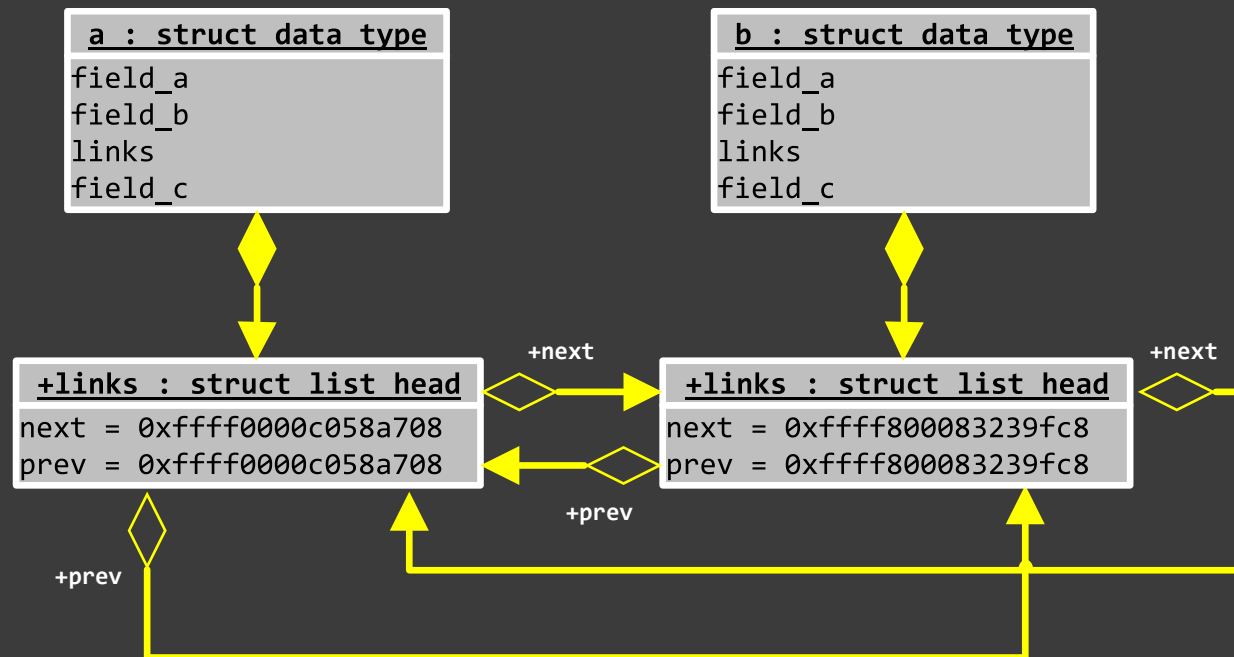


# struct list\_head



# Linked Data Structures

```
data_type
+field_a : int
+field_b : int
+links   : struct list_head
+field_c : int
```



# Exercise Adv1

- ◎ **Goal:** Learn how to navigate linked lists
- ◎ **Patterns:** [Structure Field Collection](#)
- ◎ **Memory dump:** /A64/K1 or /x64/K1
- ◎ [/AdvLCDA-Dumps/Exercise-Adv1-Linked-Lists.pdf](#)

# List Command Summary (crash)

Available in a recording and a forthcoming book

# Signs of Past Behavior

# Memory Analysis Patterns

- ◉ [Rough Stack Trace](#)
- ◉ [Past Stack Trace](#)
- ◉ Braided Stack Trace
- ◉ [Execution Residue](#)
- ◉ [Historical Information](#)
- ◉ Data Signal
- ◉ [Hidden Exception](#) / [Handled Exception](#)

# Stack Trace

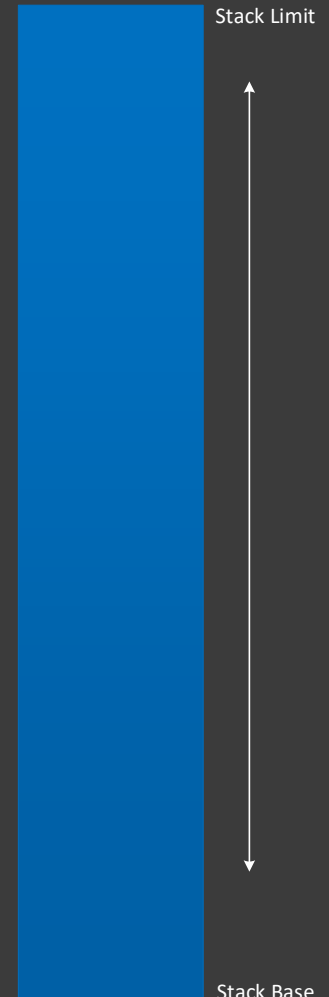
```
(gdb) bt
#0  0x000000000420174 in nanosleep ()
#1  0x000000000438e34 in sleep ()
#2  0x000000000403520 in procNE() ()
#3  0x000000000403578 in bar_two() ()
#4  0x00000000040358c in foo_two() ()
#5  0x0000000004035a4 in thread_two(void*) ()
#6  0x0000000004183f4 in start_thread ()
#7  0x00000000043dd20 in thread_start ()
```

# Execution Residue

```
(gdb) set print asm-demangle on
```

```
(gdb) x/512a $sp-2000
```

```
...  
0xffffe79bbe580: 0xffffffff      0x10000  
0xffffe79bbe590: 0x0          0x0  
0xffffe79bbe5a0: 0x0          0x0  
0xffffe79bbe5b0: 0xffffe79bbe610 0x420168 <nanosleep+24>  
0xffffe79bbe5c0: 0x0          0x0  
0xffffe79bbe5d0: 0x438e34 <sleep+272> 0xffffe79bbe650  
0xffffe79bbe5e0: 0xffffe79bbe650 0x0  
0xffffe79bbe5f0: 0x0          0x0  
0xffffe79bbe600: 0x0          0x0  
0xffffe79bbe610: 0xffffe79bbe800 0x403520 <procNE()+20>  
0xffffe79bbe620: 0xffffe79bbf070 0x0  
0xffffe79bbe630: 0x4e0000     0x403594 <thread_two(void*)>  
0xffffe79bbe640: 0x0          0x0  
0xffffe79bbe650: 0xffffffff4   0x3b985e11  
0xffffe79bbe660: 0x0          0x0  
0xffffe79bbe670: 0x0          0x0  
0xffffe79bbe680: 0x0          0x0  
0xffffe79bbe690: 0x0          0x0  
0xffffe79bbe6a0: 0x0          0x0  
...
```



# Rough Stack Trace

```
(gdb) x/512a $sp-2000
```

```
...
```

```
0xffffe79bbe350: 0xffffe79bbe360 0x403304 <work_7()+12>  
0xffffe79bbe360: 0xffffe79bbe370 0x403318 <work_6()+12>  
0xffffe79bbe370: 0xffffe79bbe380 0x40332c <work_5()+12>  
0xffffe79bbe380: 0xffffe79bbe390 0x403340 <work_4()+12>  
0xffffe79bbe390: 0xffffe79bbe3a0 0x403354 <work_3()+12>  
0xffffe79bbe3a0: 0xffffe79bbe3b0 0x403368 <work_2()+12>  
0xffffe79bbe3b0: 0xffffe79bbe3c0 0x40337c <work_1()+12>  
0xffffe79bbe3c0: 0xffffe79bbe3d0 0x403394 <work()+16>  
0xffffe79bbe3d0: 0xffffe79bbe7e0 0x40347c <procNB()+12>
```

```
...
```

```
0xffffe79bbe5b0: 0xffffe79bbe610 0x420168 <nanosleep+24>  
0xffffe79bbe5d0: 0x438e34 <sleep+272> 0xffffe79bbe650  
0xffffe79bbe610: 0xffffe79bbe800 0x403520 <procNE()+20>
```

```
...
```

```
0xffffe79bbe7f0: 0x0 0x403518 <procNE()+12>  
0xffffe79bbe800: 0xffffe79bbe810 0x403578 <bar_two()+12>  
0xffffe79bbe810: 0xffffe79bbe820 0x40358c <foo_two()+12>  
0xffffe79bbe820: 0xffffe79bbe830 0x4035a4 <thread_two(void*+16>  
0xffffe79bbe830: 0xffffe79bbe850 0x4183f4 <start_thread+180>
```

```
...
```

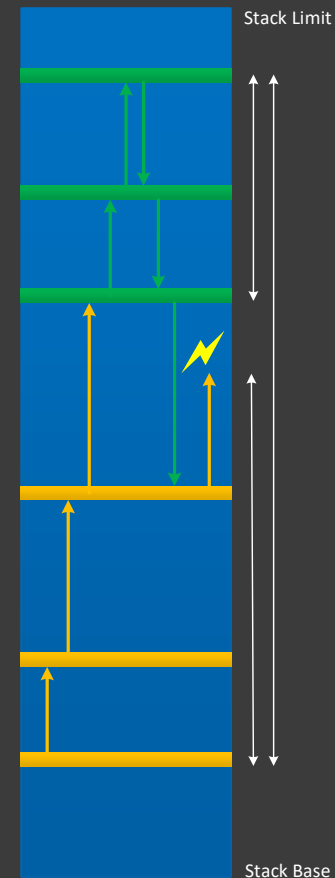
```
0xffffe79bbe850: 0x0 0x43dd20 <thread_start+48>  
0xffffe79bbe860: 0xffffe79bbf070 0x4f9540 <__default_pthread_attr>
```

```
...
```

# Past Stack Trace

```
(gdb) p $sp  
$1 = (void *) 0xffffe79bbe5d0
```

```
...  
0xffffe79bbe358: 0x403304 <work_7()+12>  
0xffffe79bbe368: 0x403318 <work_6()+12>  
0xffffe79bbe378: 0x40332c <work_5()+12>  
0xffffe79bbe388: 0x403340 <work_4()+12>  
0xffffe79bbe398: 0x403354 <work_3()+12>  
0xffffe79bbe3a8: 0x403368 <work_2()+12>  
0xffffe79bbe3b8: 0x40337c <work_1()+12>  
0xffffe79bbe3c8: 0x403394 <work()+16>  
0xffffe79bbe3d8: 0x40347c <procNB()+12>  
0xffffe79bbe5b8: 0x420168 <nanosleep+24>  
0xffffe79bbe5d0: 0x438e34 <sleep+272>  
0xffffe79bbe618: 0x403520 <procNE()+20>  
0xffffe79bbe7f8: 0x403518 <procNE()+12>  
0xffffe79bbe808: 0x403578 <bar_two()+12>  
0xffffe79bbe818: 0x40358c <foo_two()+12>  
0xffffe79bbe828: 0x4035a4 <thread_two(void*+16>  
0xffffe79bbe838: 0x4183f4 <start_thread+180>  
0xffffe79bbe858: 0x43dd20 <thread_start+48>  
0xffffe79bbe868: 0x4f9540 <__default_pthread_attr>  
...
```



# Braided Stack Trace

There can be several Past Stack Traces

...

0xffffe79bbe358: 0x403304 <work\_7()+12>

0xffffe79bbe368: 0x403318 <work\_6()+12>

0xffffe79bbe378: 0x40332c <work\_5()+12>

0xffffe79bbe388: 0x403340 <work\_4()+12>

0xffffe79bbe398: 0x403354 <work\_3()+12>

0xffffe79bbe3a8: 0x403368 <work\_2()+12>

0xffffe79bbe3b8: 0x40337c <work\_1()+12>

0xffffe79bbe3c8: 0x403394 <work()+16>

0xffffe79bbe3d8: 0x40347c <procNB()+12>

0xffffe79bbe5b8:

<nanosleep+24>

0xffffe79bbe5d0:

<sleep+272>

0xffffe79bbe618:

<procNE()+20>

0xffffe79bbe7e8: <erased>

0xffffe79bbe7f8: 0x403518 <procNE()+12>

0xffffe79bbe808:

<bar\_two()+12>

0xffffe79bbe818:

<foo\_two()+12>

0xffffe79bbe828:

<thread\_two(void\*+16>

0xffffe79bbe838:

<start\_thread+180>

0xffffe79bbe858:

<thread\_start+48>

...

# Historical Information

- Anything of forensic significance
- Used to reconstruct past behavior
- Hypothesis generation (diagnostic abduction)
- Data in memory regions
- **Example:** heap data fragments

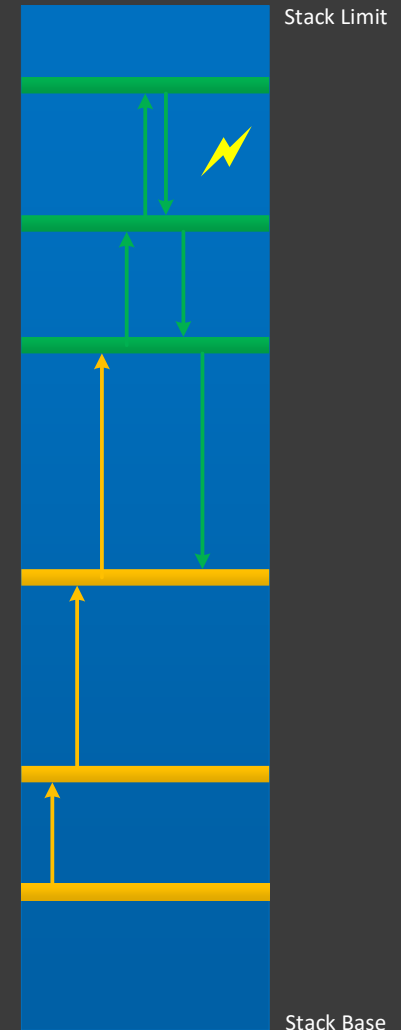
# Data Signal

0xff7668e39030:	0x0000000000000000	0x0000000000000000
0xff7668e39040:	0x0000000000000000	0x0000000000000000
0xff7668e39050:	0x0000000000000000	0x0000000000000000
0xff7668e39060:	0x0000000000000000	0x0000000000000000
0xff7668e39070:	0x0000000000000000	0x00000899000004b1
0xff7668e39080:	0x6f72702d656b6166	0x0000312d73736563
0xff7668e39090:	0x0000000000000000	0x0000000000000000
0xff7668e390a0:	0x0000000000000000	0x0000000000000000
0xff7668e390b0:	0x0000000000000000	0x0000000000000000
0xff7668e390c0:	0x0000000000000000	0x0000000000000000
0xff7668e390d0:	0x0000000000000000	0x0000000000000000
0xff7668e390e0:	0x0000000000000000	0x0000000000000000
0xff7668e390f0:	0x0000000000000000	0x0000000000000000
0xff7668e39100:	0x0000000000000000	0x0000000000000000
0xff7668e39110:	0x0000000000000000	0x0000000000000000
0xff7668e39120:	0x0000000000000000	0x0000000000000000
0xff7668e39130:	0x0000000000000000	0x0000000000000000
0xff7668e39140:	0x0000000000000000	0x0000000000000000
0xff7668e39150:	0x0000000000000000	0x0000000000000000
0xff7668e39160:	0x0000000000000000	0x0000089a000004b2
0xff7668e39170:	0x6f72702d656b6166	0x0000322d73736563
0xff7668e39180:	0x0000000000000000	0x0000000000000000
0xff7668e39190:	0x0000000000000000	0x0000000000000000
0xff7668e391a0:	0x0000000000000000	0x0000000000000000
0xff7668e391b0:	0x0000000000000000	0x0000000000000000
0xff7668e391c0:	0x0000000000000000	0x0000000000000000
0xff7668e391d0:	0x0000000000000000	0x0000000000000000
0xff7668e391e0:	0x0000000000000000	0x0000000000000000
0xff7668e391f0:	0x0000000000000000	0x0000000000000000
0xff7668e39200:	0x0000000000000000	0x0000000000000000



# Hidden/Handled Exception

```
...
0xffffe793ad1a0: 0x10000 0x810000
0xffffe793ad1b0: 0x0      0x0
0xffffe793ad1c0: 0xffffe793ad1e0  0x4145cc <_Unwind_RaiseException_Phase2+144>
0xffffe793ad1d0: 0xffffffffffffffff 0x76a28b436af36f00
0xffffe793ad1e0: 0xffffe793ad8a0  0x414bf4 <_Unwind_RaiseException+324>
0xffffe793ad1f0: 0xffffe793add30  0xffffe793ae0f0
0xffffe793ad200: 0xffffe74000b80  0xffffe793ad970
0xffffe793ad210: 0x0      0xffffe793ae770
0xffffe793ad220: 0x0      0x0
0xffffe793ad230: 0x0      0x0
...
0xffffe793adc30: 0x0      0x0
0xffffe793adc40: 0x0      0x0
0xffffe793adc50: 0x0      0x0
0xffffe793adc60: 0x0      0x0
0xffffe793adc70: 0x0      0x0
0xffffe793adc80: 0xffffe793ae770  0x404d7c <__cxa_throw+144>
0xffffe793adc90: 0x0      0x0
0xffffe793adca0: 0x0      0x414ab0 <_Unwind_RaiseException>
0xffffe793adcb0: 0x4000000000000000  0x0
0xffffe793adcc0: 0x0      0x0
0xffffe793adcd0: 0x0      0x0
...
```



# GDB Extensions

# The Need for Extensions

- ⦿ Limitations of existing commands
- ⦿ Scripts may be slow or not convenient to use
- ⦿ Different output format
- ⦿ Get more insight
- ⦿ Automate analysis of verbose output

# Stack Log

## Stack trace/backtrace/rough stack trace

```
0xffffe79bbe358: 0x403304 <work_7()+12>  
0xffffe79bbe368: 0x403318 <work_6()+12>  
0xffffe79bbe378: 0x40332c <work_5()+12>  
0xffffe79bbe388: 0x403340 <work_4()+12>  
0xffffe79bbe398: 0x403354 <work_3()+12>  
0xffffe79bbe3a8: 0x403368 <work_2()+12>  
0xffffe79bbe3b8: 0x40337c <work_1()+12>  
0xffffe79bbe3c8: 0x403394 <work()+16>  
0xffffe79bbe3d8: 0x40347c <procNB()+12>  
0xffffe79bbe4a8: 0x438e08 <sleep+228>  
0xffffe79bbe578: 0x438e28 <sleep+260>  
0xffffe79bbe5b8: 0x420168 <nanosleep+24>
```

## Stack log

**stack\_address,return\_address,module,symbol,offset**

```
0xffffe79bbe358,0x403304,/mnt/c/ALCDA2/A64/App8/App8,work_7,12  
0xffffe79bbe368,0x403318,/mnt/c/ALCDA2/A64/App8/App8,work_6,12  
0xffffe79bbe378,0x40332c,/mnt/c/ALCDA2/A64/App8/App8,work_5,12  
0xffffe79bbe388,0x403340,/mnt/c/ALCDA2/A64/App8/App8,work_4,12  
0xffffe79bbe398,0x403354,/mnt/c/ALCDA2/A64/App8/App8,work_3,12  
0xffffe79bbe3a8,0x403368,/mnt/c/ALCDA2/A64/App8/App8,work_2,12  
0xffffe79bbe3b8,0x40337c,/mnt/c/ALCDA2/A64/App8/App8,work_1,12  
0xffffe79bbe3c8,0x403394,/mnt/c/ALCDA2/A64/App8/App8,work,16  
0xffffe79bbe3d8,0x40347c,/mnt/c/ALCDA2/A64/App8/App8,procNB,12  
0xffffe79bbe4a8,0x438e08,/mnt/c/ALCDA2/A64/App8/App8,sleep,228  
0xffffe79bbe578,0x438e28,/mnt/c/ALCDA2/A64/App8/App8,sleep,260  
0xffffe79bbe5b8,0x420168,/mnt/c/ALCDA2/A64/App8/App8,nanosleep,24
```

# Types of GDB Extensions

- ⦿ GDB internal language: not enough string processing
- ⦿ Python language
- ⦿ Automation from outside

# Exercise Adv2

- ◎ **Goal:** Create GDB extension in Python
- ◎ **Patterns:** [Execution Residue](#); [Rough Stack Trace](#); [Past Stack Trace](#); Stack Log
- ◎ **Memory dump:** /A64/App8 or /x64/App8
- ◎ [/AdvLCDA-Dumps/Exercise-Adv2-GDB-Extensions.pdf](#)

# Exercise Adv3

- ◎ **Goal:** Create GDB extension in native language
- ◎ **Patterns:** [Execution Residue](#); [Rough Stack Trace](#); [Past Stack Trace](#); Stack Log
- ◎ **Memory dump:** /A64/App8 or /x64/App8
- ◎ [/AdvLCDA-Dumps/Exercise-Adv3-GDB-Extensions.pdf](#)

# Used Command Summary (GDB)

Available in a recording and a forthcoming book

# Script Problems to Watch

- ① Unwanted terminology
- ① Unwanted sorting
- ① Unwanted deduplication

# Homework

- ⦿ Verify for symbols from shared libraries
- ⦿ Add a column for Coincidental Symbolic Information indicator (true return address)
- ⦿ Stack log for backtrace
- ⦿ How would you approach automation of the **Braided Stack Trace** analysis pattern?

# LLM Analysis Assistants

# Analysis with LLM

- ⦿ Helps with analysis
- ⦿ Makes analysis more complex
- ⦿ Split tasks into well-defined individual scripts
- ⦿ Ask for command line parameters

# Notes on Coding Assistants

# Modeling with LLM

- ① Determine diagnostic indicators for a problem
- ② Ask to write a program that models a problem and diagnostic indicators
- ③ Explore different mechanisms that end with the same problem and diagnostic indicators

# Shared Memory Leak

```
...  
0000ff7668e39000 121130 66864 32 rw-s- [ anon ]  
0000ff7668e3a000 121130 66864 32 rw-s- [ anon ]  
0000ff7668e3b000 121130 69168 32 rw-s- [ anon ]  
0000ff7668e3c000 121130 67696 32 rw-s- [ anon ]  
...
```

```
...  
0xff7668e39030: 0x0000000000000000 0x0000000000000000  
0xff7668e39040: 0x0000000000000000 0x0000000000000000  
0xff7668e39050: 0x0000000000000000 0x0000000000000000  
0xff7668e39060: 0x0000000000000000 0x0000000000000000  
0xff7668e39070: 0x0000000000000000 0x00000899000004b1  
0xff7668e39080: 0x6f72702d656b6166 0x0000312d73736563  
0xff7668e39090: 0x0000000000000000 0x0000000000000000  
0xff7668e390a0: 0x0000000000000000 0x0000000000000000  
0xff7668e390b0: 0x0000000000000000 0x0000000000000000  
0xff7668e390c0: 0x0000000000000000 0x0000000000000000  
0xff7668e390d0: 0x0000000000000000 0x0000000000000000  
...
```

# Exercise Adv4

- ◎ **Goal:** Model shared memory leak
- ◎ **Patterns:** [Paratext](#); Data Signal
- ◎ </AdvLCDA-Dumps/Exercise-Adv4-Model-shared-memory-leak.pdf>

# Used Command Summary (Mix)

Available in a recording and a forthcoming book

# Braided Stack Trace



# Exercise Adv5

- ◎ **Goal:** Model braided stack trace
- ◎ **Patterns:** Braided Stack Trace
- ◎ </AdvLCDA-Dumps/Exercise-Adv5-Model-braided-stack-trace.pdf>

# Call Trace and Paths (Strands)

```
// function_name,local_size
```

```
foo00,64
  foo11,220
    foo12,3620
  foo21,420
    foo22,500
      foo23,560
  foo31,310
    foo32,600
      foo33,1064
        foo43,128
          foo44,280
```

foo00 → foo11 → foo12

foo00 → foo21 → foo22 → foo23

foo00 → foo31 → foo32 → foo33

foo00 → foo31 → foo32 → foo43 → foo44

# Braided Stack Trace Results



# Used Command Summary (Mix)

Available in a recording and a forthcoming book

# Lessons Learned

- ① Stack growth direction
- ① Top and bottom
- ① Learn when to stop

# Homework

- ⦿ Model a situation when there are no crossings
- ⦿ What would be an ideal **Braided Stack Trace** picture if we only use a call trace diagram (no region dump)? Model this case

# N-Pointer Dereference

## ◎ Pointer

0xffffe79bbe360: 0x0000ffffe79bbe370

## ◎ Dereference

0xffffe79bbe370: 0x0000ffffe79bbe380

## ◎ Double dereference

0xffffe79bbe380: 0x0000ffffe79bbe390

## ◎ Triple dereference

0xffffe79bbe390: 0x0000ffffe79bbe3a0

# Memory Analysis Patterns

## ◎ Pointer Orbit

0xffffe79bbe360: 0x0000fffe79bbe370 0x0000fffe79bbe380 0x0000fffe79bbe390

## ◎ Region Brane

0xffffe79bbe360: 0x0000fffe79bbe370 0x0000fffe79bbe380 0x0000fffe79bbe390

0xffffe79bbe368: 0x0000000000403318 0xd65f03c0a8c17bfd

0xffffe79bbe370: 0x0000fffe79bbe380 0x0000fffe79bbe390 0x0000fffe79bbe3a0

## ◎ Region Orbit

[0xffffe79bbe360-0xffffe79bbe378): {0x0000fffe79bbe370, 0x0000fffe79bbe380  
0x0000fffe79bbe390, 0x0000fffe79bbe3a0, 0x0000000000403318  
0xd65f03c0a8c17bfd}

# Classic GDB Script (human)

```
define dpp
  set $i = 0
  set $p = $arg0
  while $i < $arg1
    printf "%p: ", $p
    x/ga *(long *)$p
    set $i = $i + 1
    set $p = $p + 8
  end
end
```

```
(gdb) dpp 0xffffe79bbe360 3
```

```
0xffffe79bbe360: 0xffffe79bbe370: 0xffffe79bbe380
```

```
0xffffe79bbe368: 0x403318 <_Z6work_6v+12>: 0xd65f03c0a8c17bfd
```

```
0xffffe79bbe370: 0xffffe79bbe380: 0xffffe79bbe390
```

# Problems

- Only 2 levels
- Colons for all values
- Stops on memory access error
- Mangled symbols by default
- No file output

# Exercise Adv6

- ◎ **Goal:** Create N-dereferencing GDB extension in Python
- ◎ **Patterns:** [Execution Residue](#)
- ◎ **Memory dump:** /A64/App8 or /x64/App8
- ◎ [/AdvLCDA-Dumps/Exercise-Adv6-N-Dereferencing.pdf](#)

# Used Command Summary (GDB)

Available in a recording and a forthcoming book

# Refinement Lessons Learned

- ⦿ Do not quit GDB on error
- ⦿ Do not stop on memory access errors (use Python scripts)
- ⦿ Optionally exclude symbols
- ⦿ Only output colons for the first addresses
- ⦿ Include CSV output
- ⦿ Remove duplicates in CSV

# Homework

- ⦿ Remove memory error messages
- ⦿ Add **Region Orbit** output
- ⦿ Add closure: iterate until memory error or cycle
- ⦿ Add highlighting of a specific value
- ⦿ Make symbol offsets a separate CSV column

# GenAI Chat Analysis Assistants

# Memory Analysis Patterns

- ◎ Annotated Stack Trace
- ◎ Disassembly Summary
- ◎ Region Summary
- ◎ Analysis Summary
- ◎ LLM can suggest new analysis patterns

# Prompt Analysis

Explain this stack trace fragment line by line: ...

Categorize this stack trace: ...

Split this stack trace into appropriate categories: ...

Summarize this disassembly: ...

Summarize this region of memory: ... Explain all symbolic references

Summarize this analysis: ...

Write the best prompt to annotate stack traces

Brutally assess the annotation and score it

Annotate 10/10

Provide your chain-of-thought for ...

# Warning

Because of the evolving nature of Generative AI LLMs and their differences, the following exercise output may differ from what you get when reproducing the results.

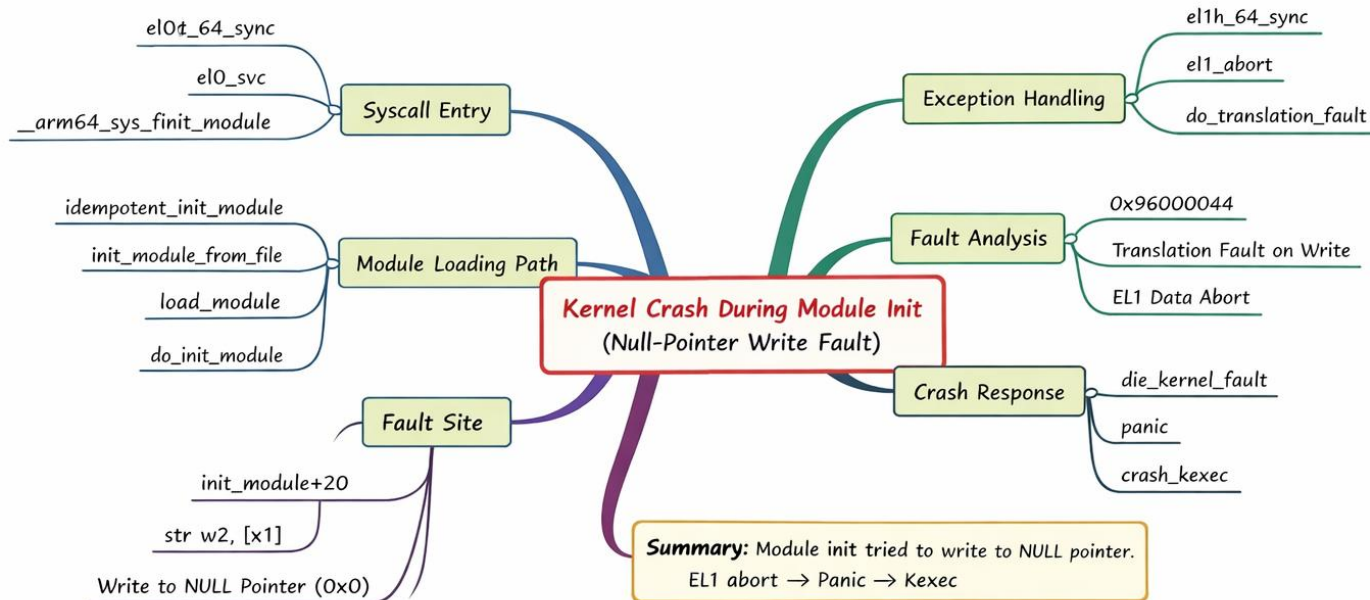
# Exercise Adv7

- ◎ **Goal:** Use a GenAI chat as a memory dump analysis assistant
- ◎ **Patterns:** [Annotated Stack Trace](#); [Exception Stack Trace](#); [Stack Trace Motif](#); Causal Stack Trace; [Disassembly Summary](#); [Region Summary](#); [Coincidental Symbolic Information](#); [Design Value](#); [Analysis Summary](#)
- ◎ </AdvLCDA-Dumps/Exercise-Adv7-GenAI-chat-assistant.pdf>

# Command Summary (crash)

Available in a recording and a forthcoming book

# Problem Mind Maps



# HL User Space Fault Simulation

```
#include <stdint.h>
#include <stdio.h>

static int init_module_like(void) {
    uint32_t *p = 0;
    uint32_t v = 1;
    return (*p = v), 0;    // deliberate null-pointer write
}

int main(void) {
    puts("About to simulate module-init null write in user space...");
    return init_module_like();
}
```

# LL User Space Fault Simulation

```
#include <stdint.h>

__attribute__((noinline))
int init_module_like(void) {
    register uint64_t x1 = 0;
    register uint32_t w2 = 1;
    register uint32_t w0 = 0;
    *(volatile uint32_t *)x1 = w2;
    return w0;
}

int main(void) {
    return init_module_like();
}
```

# Kernel Space Fault Simulation

```
// ...

static int __init safe_init_module(void)
{
    u32 *p = NULL;
    u32 v = 1;

    pr_info("safe_null_sim: init entered\n");
    pr_info("safe_null_sim: would execute store of %u through pointer %px\n", v, p);

    if (!p) {
        pr_err("safe_null_sim: simulated fault avoided: null-pointer write blocked\n");
        return -EFAULT;
    }

    *p = v;
    return 0;
}

// ...
```

# Full Crash Path Simulation

```
// ...

/* Safe simulation of the faulting instruction */
static bool simulated_store_u32(uint32_t *p, uint32_t value, struct fault_context *ctx)
{
    (void)value;

    if (p == NULL) {
        ctx->pc_symbol = "init_module+20";
        ctx->fault_addr = 0x0;
        ctx->esr = 0x96000044; /* simulated: same-EL data abort, write, translation fault */
        ctx->is_write = true;
        ctx->same_el = true;
        return false;
    }

    *p = value;
    return true;
}

// ...
```

# Lessons Learned

- ① Have your own chain-of-thought (analysis patterns)
- ① Refine prompts
- ① Double-check everything you don't know that pertains to your own chain of thought

# Data Science and Visualization

# Memory Analysis Patterns

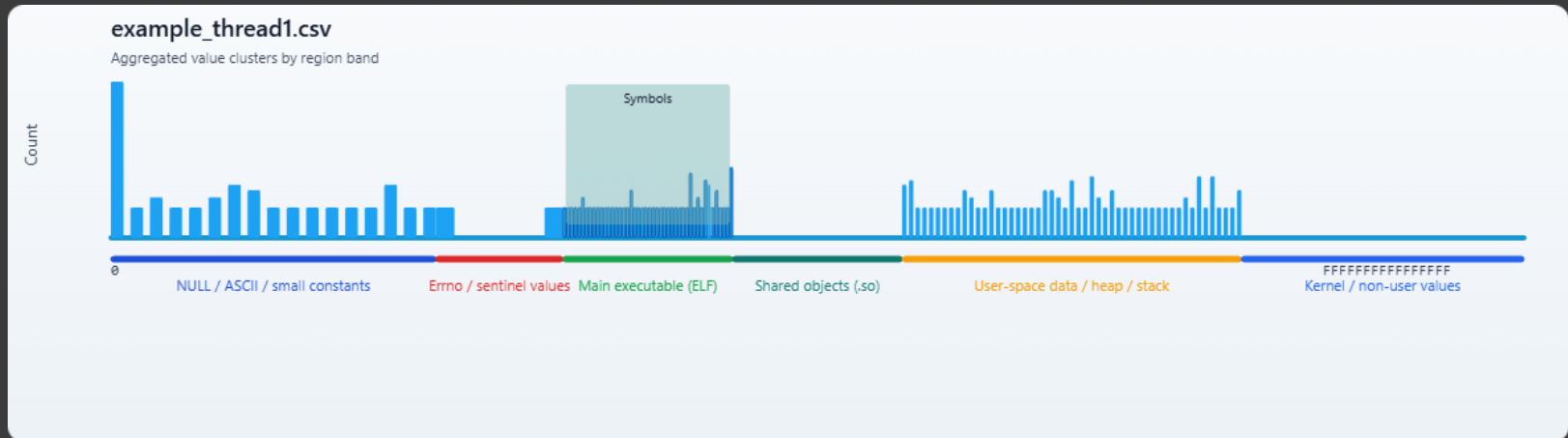
- ◎ Region Profile
- ◎ Region Clusters
- ◎ Region Spectrum
- ◎ Normalized Region



# Region Clusters

Input: address, value, symbol

Output: value, symbol, count, band



# Region Spectrum

Input: address, value, symbol

Output: address, value, symbol, class

## example\_thread4.csv

Region spectrum by address order. Zero values are rendered as blank gaps.

Module   Heap   Stack   Foreign Stack   Small Value   Out-of-range   Error



# Normalized Region

Input: address, value, symbol

Output: normalized\_address, normalized\_value, symbol [, annotations]

**address,value,symbol,original\_address,original\_value,address\_normalized,value\_normalized**

```
...  
0x00000000000000a0,0x00000000000011b1b,,0x0000ffffe7a3cde90,0x00000000000011b1b,true,false  
0x00000000000000a8,0x0000000000000000,,0x0000ffffe7a3cde98,0x0000000000000000,true,false  
0x00000000000000b0,0x00000000000000900,,0x0000ffffe7a3cdea0,0x0000ffffe7a3ce6f0,true,true  
0x00000000000000b8,0x000000000000404ddc,__cxa_rethrow+0x54,0x0000ffffe7a3cdea8,0x000000000000404ddc,true,false  
0x00000000000000c0,0x0000ffffe6c000b80,,0x0000ffffe7a3cdeb0,0x0000ffffe6c000b80,true,false  
0x00000000000000c8,0x0000000000000002,,0x0000ffffe7a3cdeb8,0x0000000000000002,true,false  
0x00000000000000d0,0x00000000fffffffff,0x0000ffffe7a3cdec0,0x00000000fffffffff,true,false  
...
```

# Exercise Adv8

- ◎ **Goal:** Explore various execution residue statistics and visualization opportunities
- ◎ **Memory Analysis Patterns:** [Execution Residue](#); [Region Profile](#); [Region Clusters](#); [Normalized Region](#)
- ◎ [/AdvLCDA-Dumps/Exercise-Adv8-data-science-visualization.pdf](#)

# Used Command Summary (Mix)

Available in a recording and a forthcoming book

# Stack Anomaly Detection

- Find if stacks differ between two runs
- Two core dumps from two runs
- Get all stack region data from both
- Normalize stack regions
- Compare stack regions

# Exercise Adv9

- ◎ **Goal:** Find anomalies in stack traces from different runs
- ◎ **Memory Analysis Patterns:** [Execution Residue](#); [Rough Stack Trace](#)
- ◎ </AdvLCDA-Dumps/Exercise-Adv9-anomalies-baseline.pdf>

# Used Command Summary (Mix)

Available in a recording and a forthcoming book

# Refinement Lessons Learned

- ⦿ Analysis pattern description and original graphics in a prompt
- ⦿ Subtitle, label, and histogram bar alignment
- ⦿ An assistant may suggest useful functionality and corresponding CLI parameters
- ⦿ CLI scripts that accept core dumps as a parameter, internally contain GDB scripts, and launch GDB

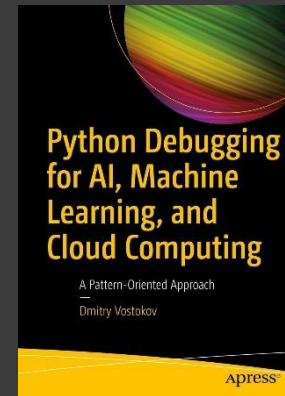
# Homework

- ⦿ Prompt for other types of visualization
- ⦿ Adapt to your hardware platform
- ⦿ Detect Foreign Stack values

# Diagnostics Presentation Patterns

- Introduced in:

[Pattern-Oriented Debugging Process](#)

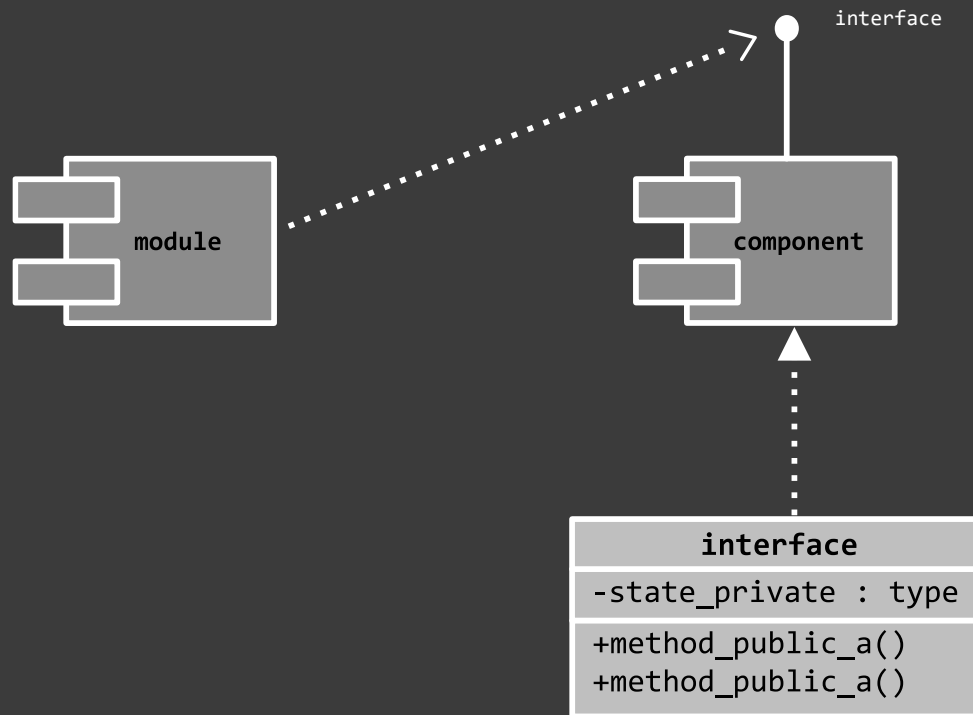


- Include visualization
- New forthcoming pattern catalog

# A Crash **Dump** Course in Unified Modeling Language

## Part II

# Components and Interfaces



# Classes and Objects (C++)

```
class interface // it can also be a struct interface
{
public:
    void method();
private:
    int state;
};
```

```
interface object_a, object_b;
```

```
void interface::method(/* interface* this */)
{
    state = 0; // this->state = 0;
}
```

```
void foo()
{
    object_a.method();
}
```

# Objects and Methods (C)

```
struct interface
{
    int state;
};

void method(struct interface* obj);

struct interface object_a, object_b;

void method(struct interface* obj)
{
    obj->state = 0;
}

void foo()
{
    method(&object_a);
}
```

# Classes and Objects Analogy (C)

```
struct object
{
    int state;
};
```

```
struct interface
{
    void (*method)(struct object* obj);
};
```

```
void method(struct object* obj)
{
    obj->state = 0;
}
```

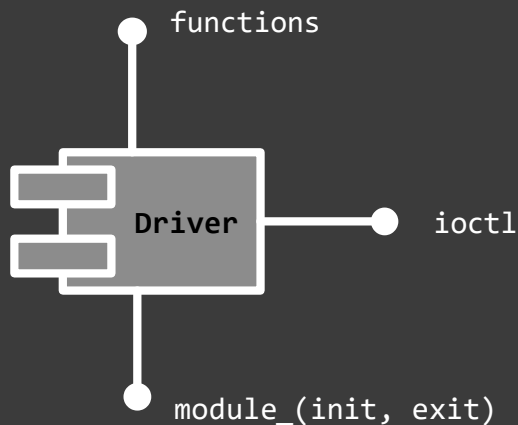
```
struct interface class_a = {&method}; struct object object_a, object_b;
```

```
void foo()
{
    class_a.method(&object_a);
}
```

# A Crash **Dump** Course in Linux Internals

# Device Driver

- A pluggable kernel component (module) for a device or several devices
- Creates device objects and registers them
- Provides entry points for I/O operations including `ioctl` interface (I/O control, used for device-specific commands)
- Implemented as C structures with data and pointers to functions
- Class analogy

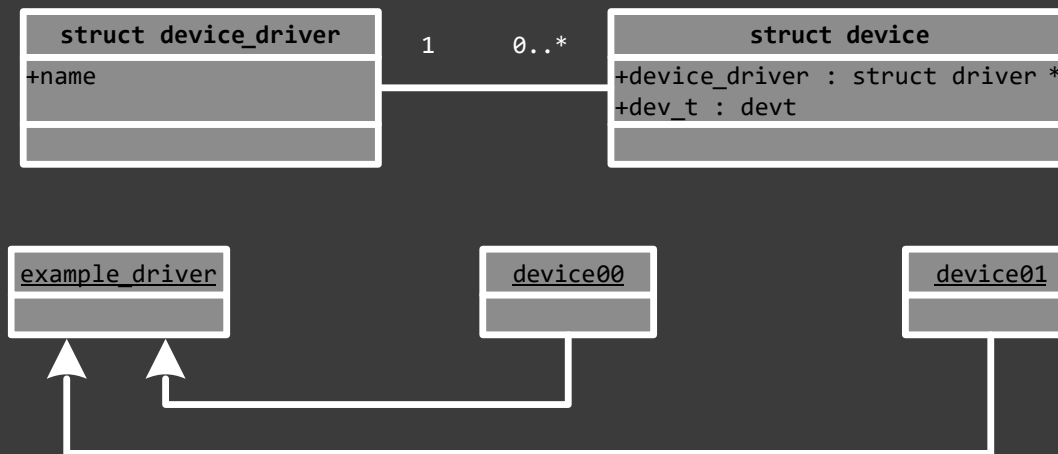


`/sys/module`

`/sys/class/<class>/<name>/device/driver`

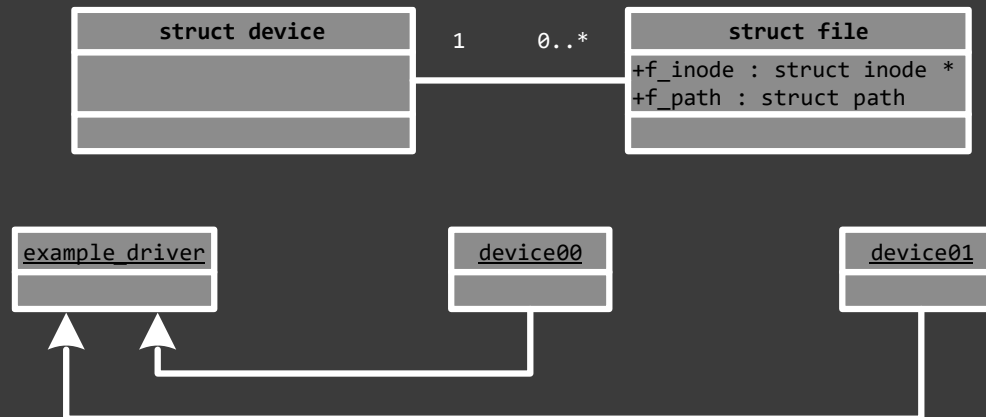
# Device

- Represents a physical or logical device in the kernel
- Target of an I/O operation, usually through an open file such as `/dev/<name>`
- Names can be found in: `/sys/devices`, `/sys/class`, `/dev`
- Implemented as a C structure
- Object analogy



# File

- Represents an open file object in the kernel
- Created when a process opens a file, device, socket, ...
- Used as the handle for I/O operations such as `read`, `write`, and `ioctl`
- Contains state for one open instance, including file position, flags, and a pointer to `file_operations`
- Implemented as a C structure
- Handle analogy



# Exercise Adv10

- ◎ **Goal:** Learn how to inspect kernel modules, drivers, devices, and files
- ◎ **Memory Analysis Patterns:** [Module Variable](#)
- ◎ </AdvLCDA-Dumps/Exercise-Adv10-kernel-modules.pdf>

# Command Summary I (crash)

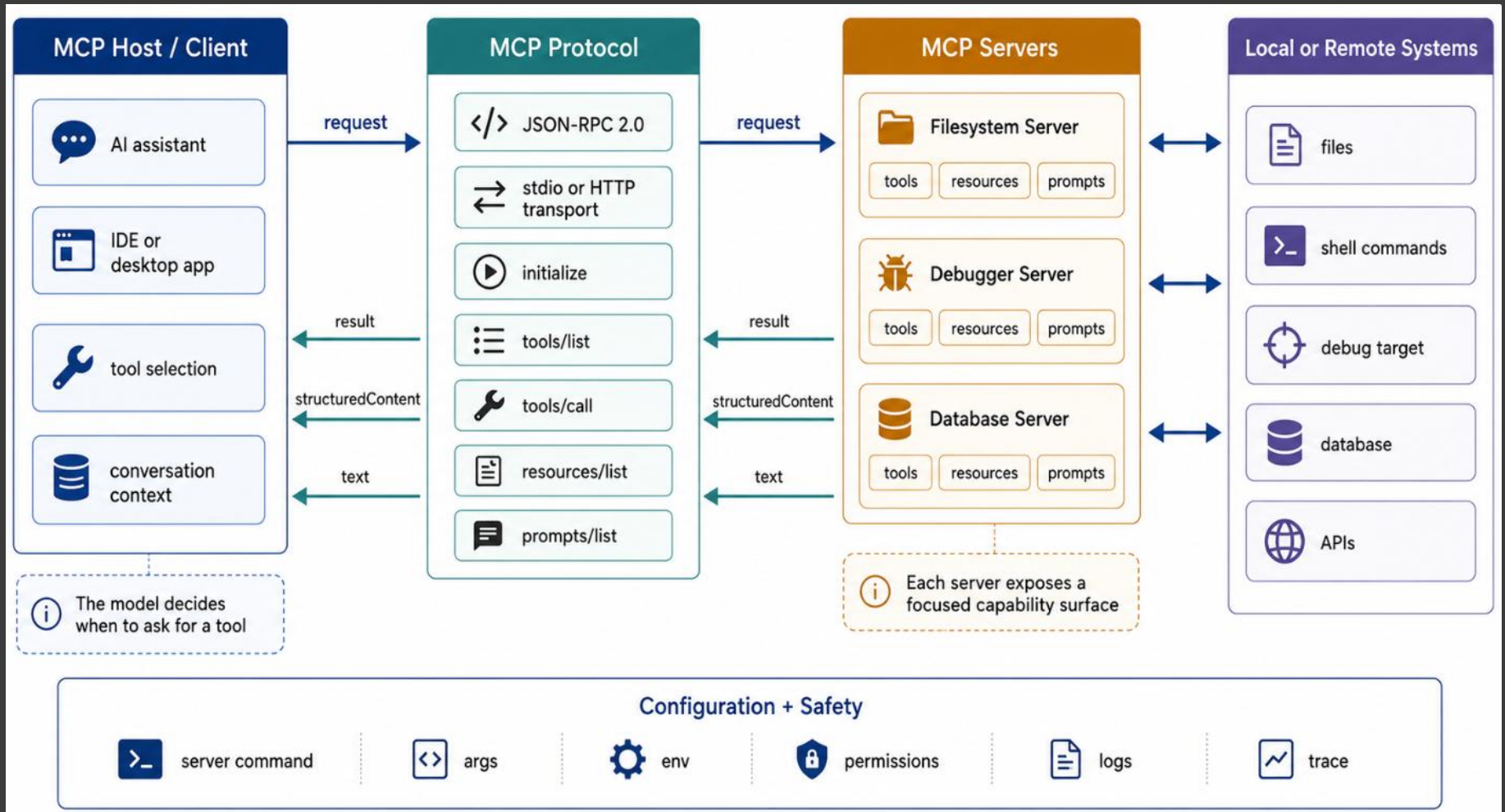
Available in a recording and a forthcoming book

# Command Summary II (crash)

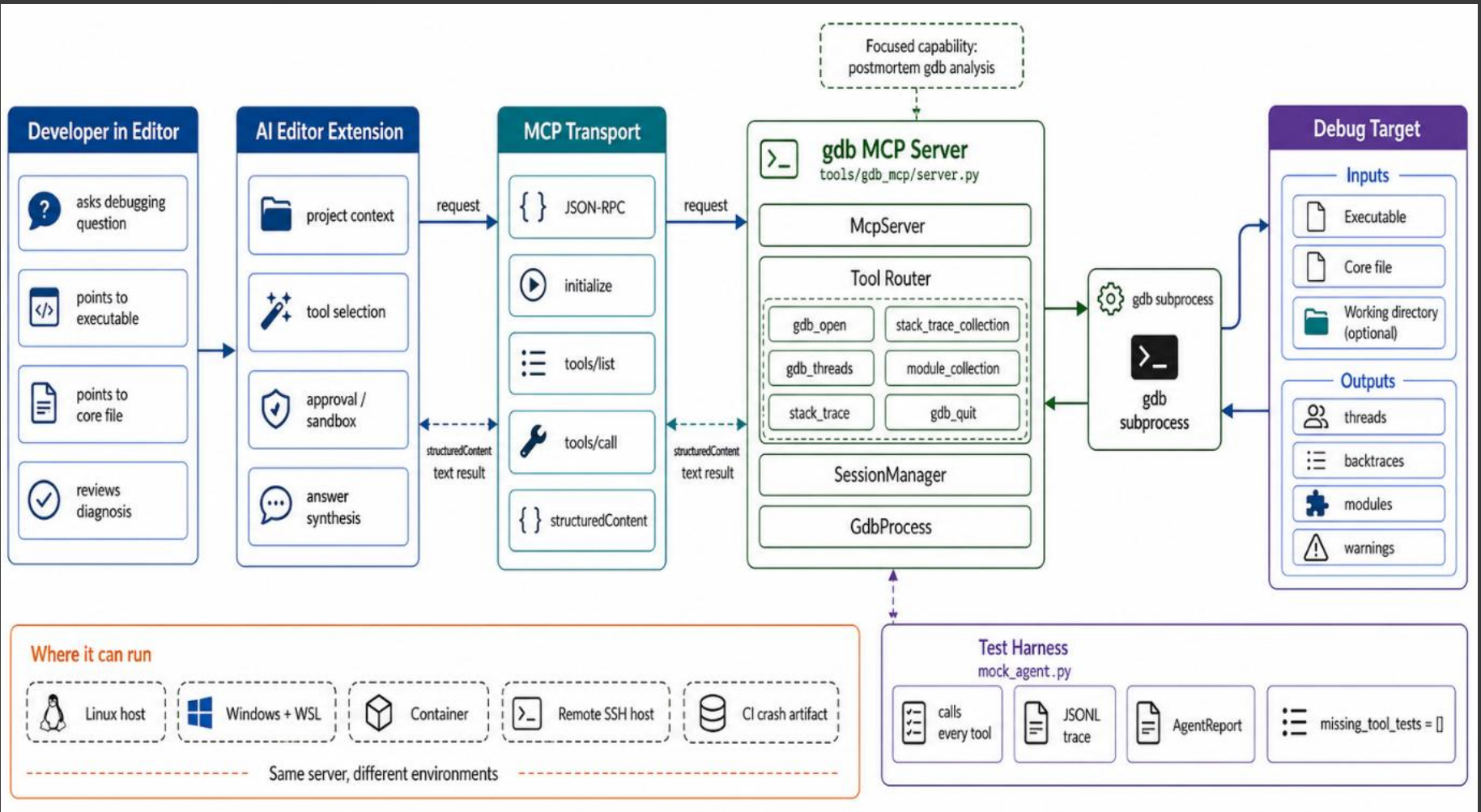
Available in a recording and a forthcoming book

# Analysis Patterns as Tools

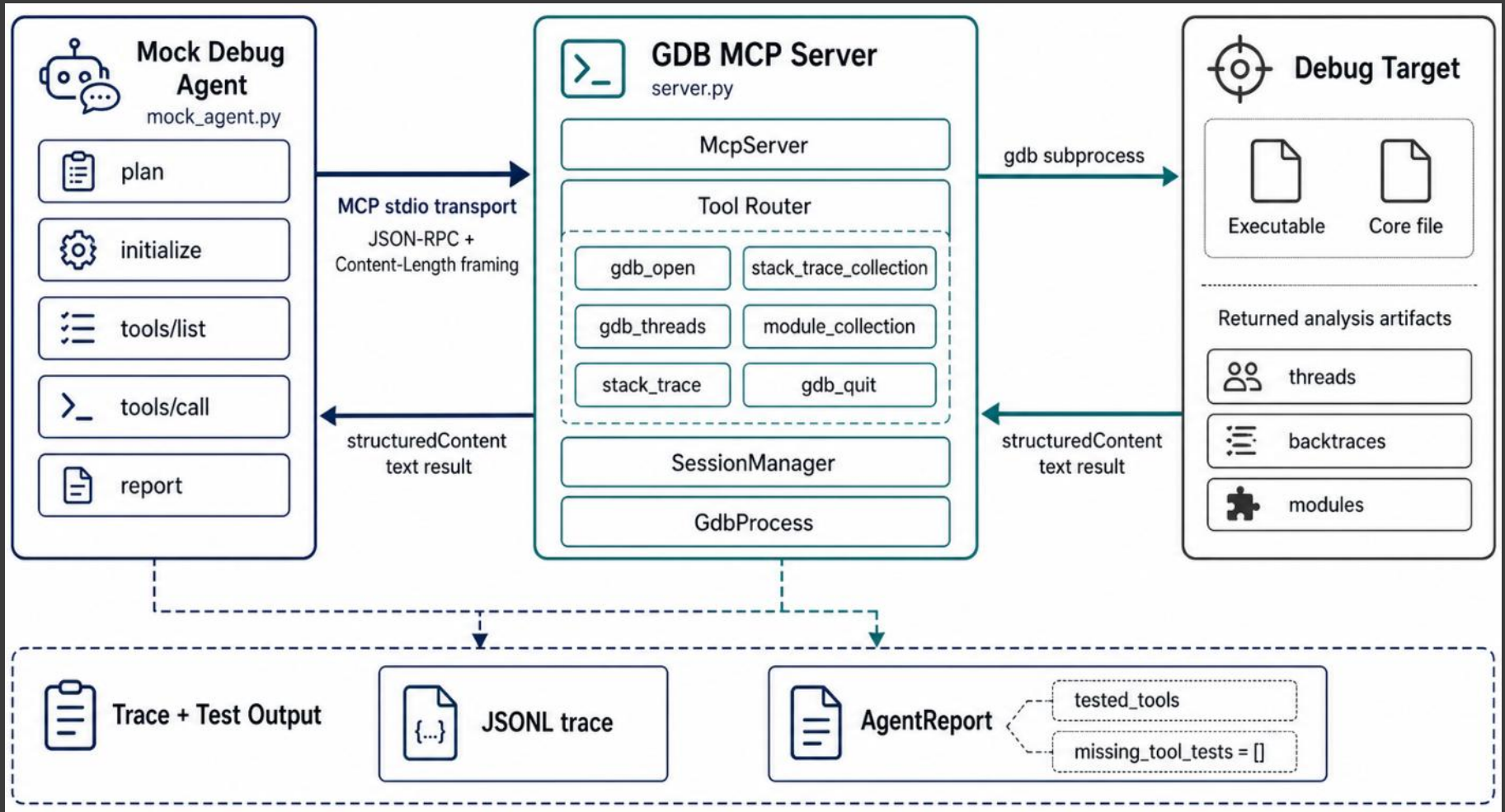
# Model Context Protocol (MCP)



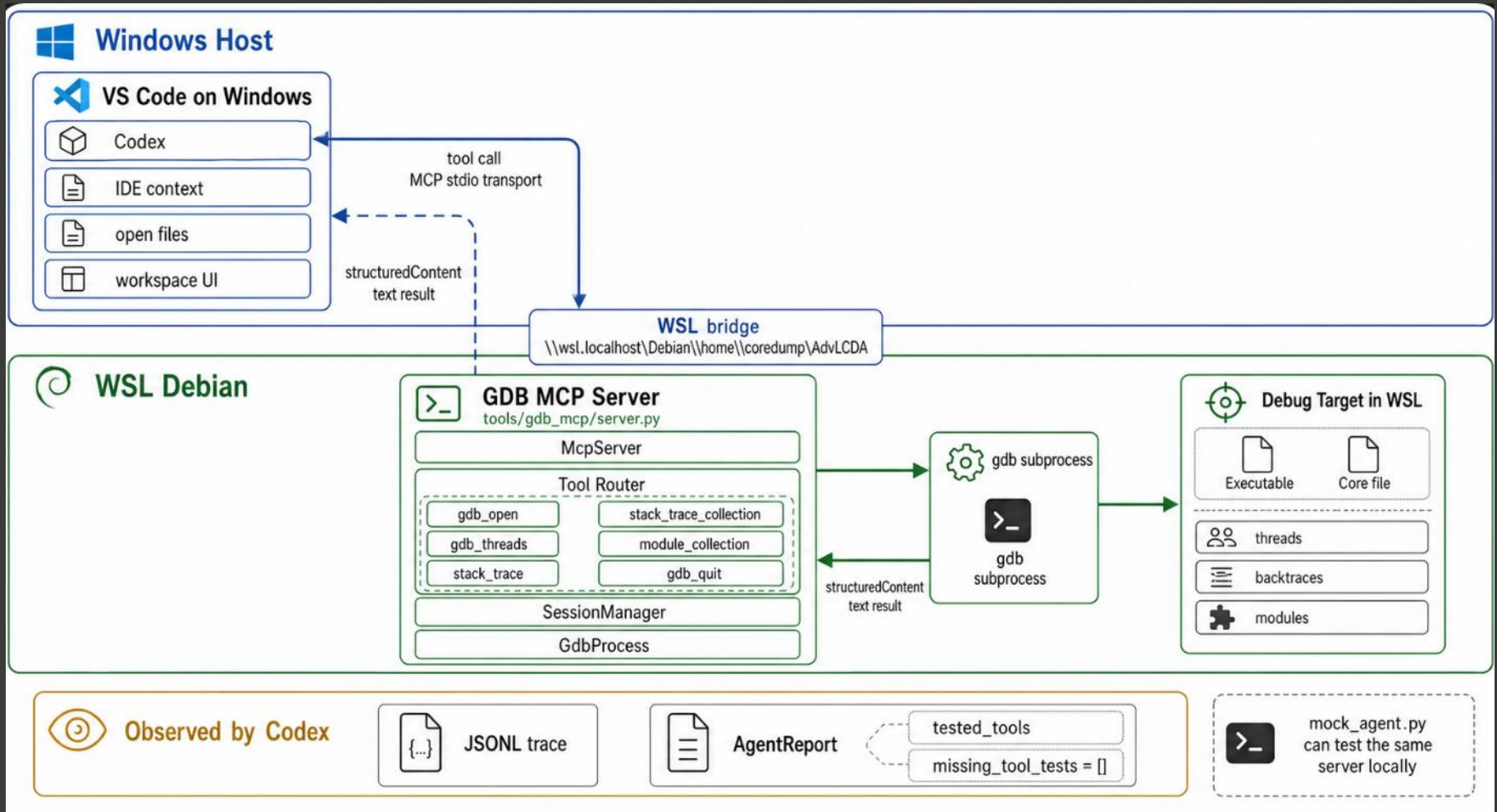
# GDB MCP Server



# Testing: AI Agent Mock



# Codex Agent



# Configuration

```
C:\Users\<<name>\.codex\config.toml
```

```
[mcp_servers.gdb-wsl]
```

```
command = "C:\\Windows\\System32\\wsl.exe"
```

```
args = ["-d", "Debian", "--cd", "/home/coredump/AdvLCDA", "--exec",  
"python3", "-u", "/home/coredump/AdvLCDA/tools/gdb_mcp/server.py"]
```

```
startup_timeout_sec = 30
```

```
tool_timeout_sec = 120
```

```
required = true
```

# Exercise Adv11

- ◎ **Goal:** Learn how GDB can be used as an MCP server with analysis patterns as tools
- ◎ **Memory Analysis Patterns:** Stack Trace Collection
- ◎ </AdvLCDA-Dumps/Exercise-Adv11-GDB-MCP.pdf>

# Command and Prompt Summary

Available in a recording and a forthcoming book

# Refinement Lessons Learned

- ⦿ `required = true` helped with debugging
- ⦿ Content framing mismatch
- ⦿ All debugging was done by Codex
- ⦿ w/o FastMCP initially

# Further Reading

[Diagnostic Analysis Patterns as MCP Tools for Agentic Diagnostics](#)

# Homework

- ⦿ Test with your favorite AI agent
- ⦿ Extend with other tools
- ⦿ Test FastMCP version (`server_fastmcp.py`)
- ⦿ Add more intelligence

# Q&A

Please send your feedback using the contact form on [PatternDiagnostics.com](https://PatternDiagnostics.com)

Thank you for attendance!