
# Intel® Inspector XE 2016

**Memory and thread debugger**

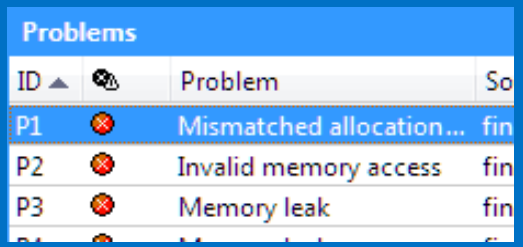
# Agenda

- Intro to Intel® Inspector XE
- Analysis workflow
- Memory problem analysis
- Threading problem Analysis
- Integration with debugger
- Automated regression testing and user API

# Intro to Intel® Inspector XE

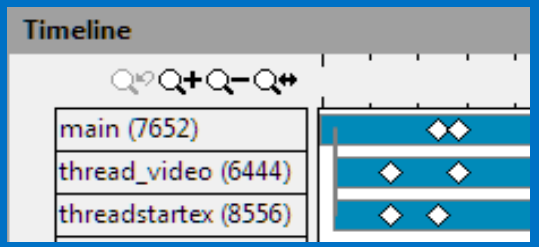
# Motivation for The Inspector XE

## Memory Errors




- Invalid Accesses
- Memory Leaks
- Uninitialized Memory Accesses

## Threading Errors



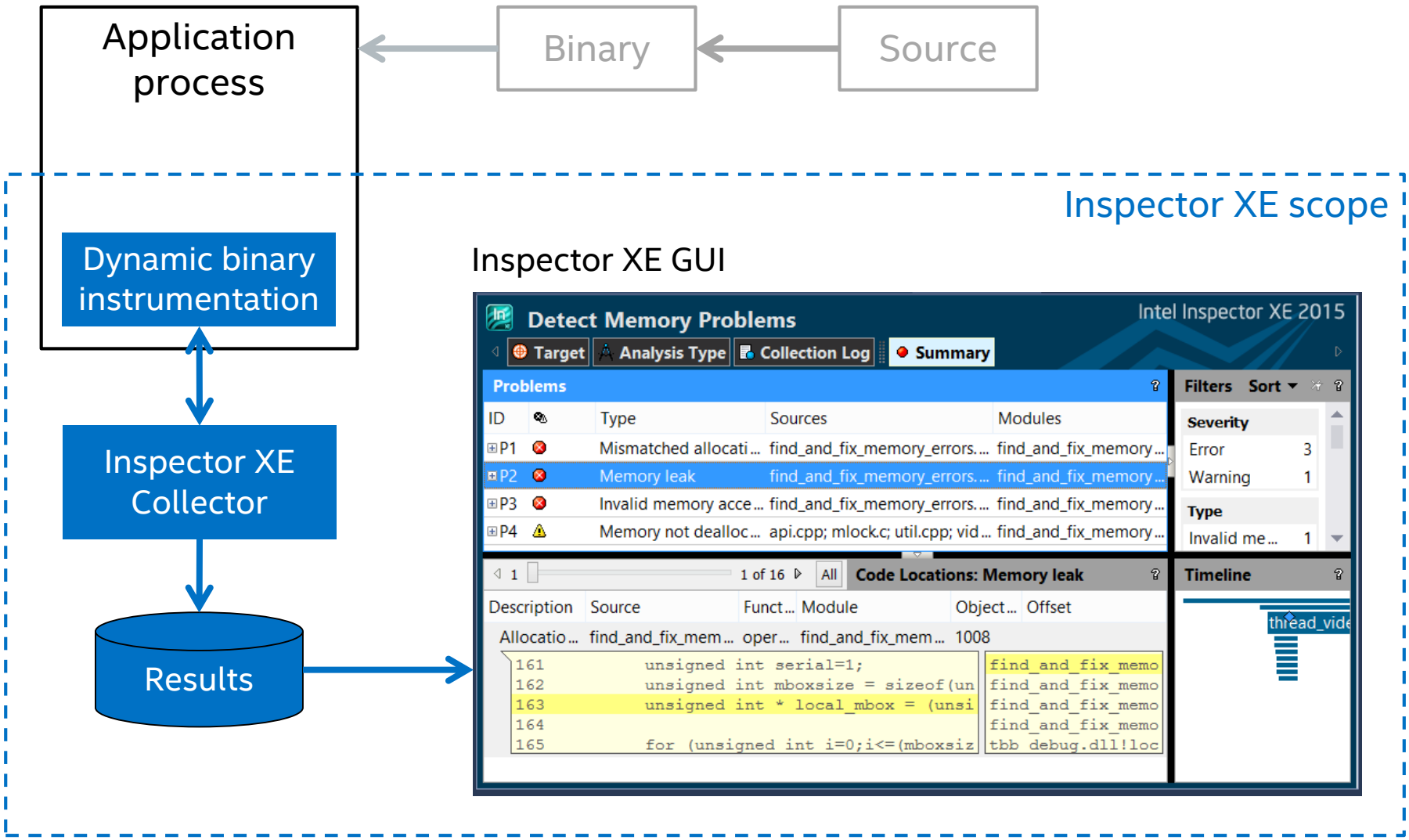

- Data Races
- Deadlocks
- Cross Stack References

## Multi-threading problems

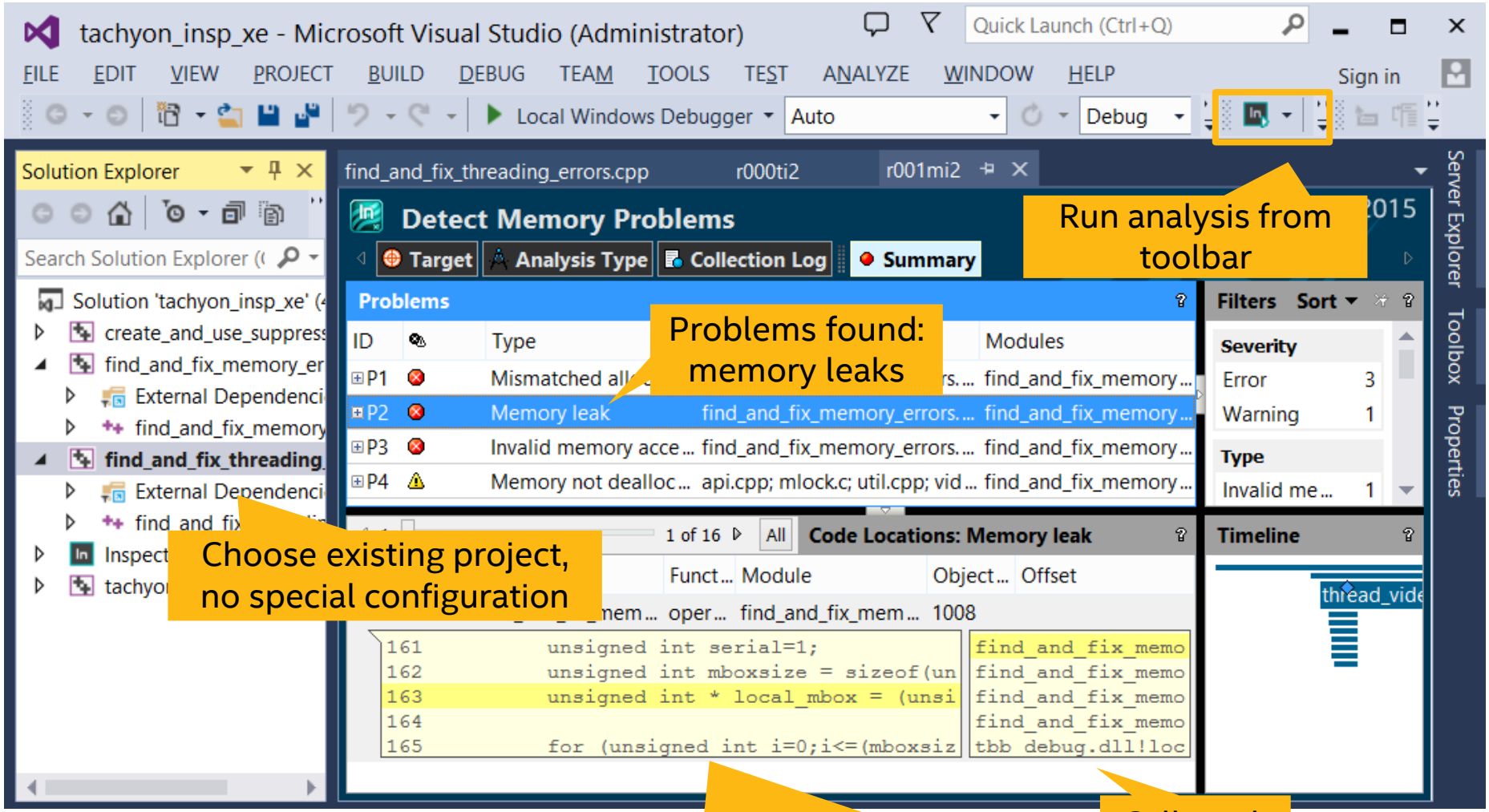
- Hard to reproduce,
- Difficult to debug
- Expensive to fix

→ Let the tool do it for you

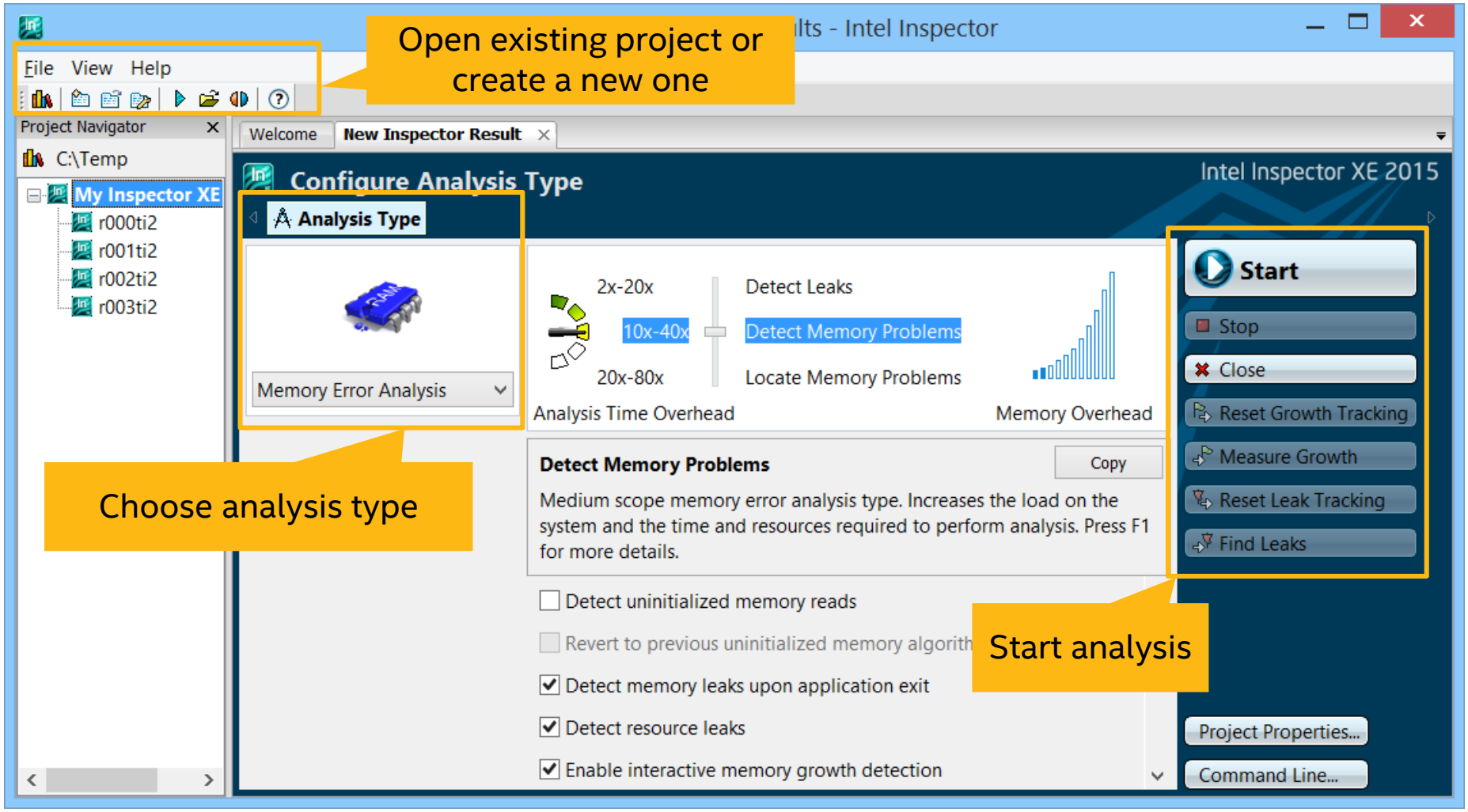
# Intel Inspector XE: Dynamic analysis



# How it looks: Visual Studio* Integration

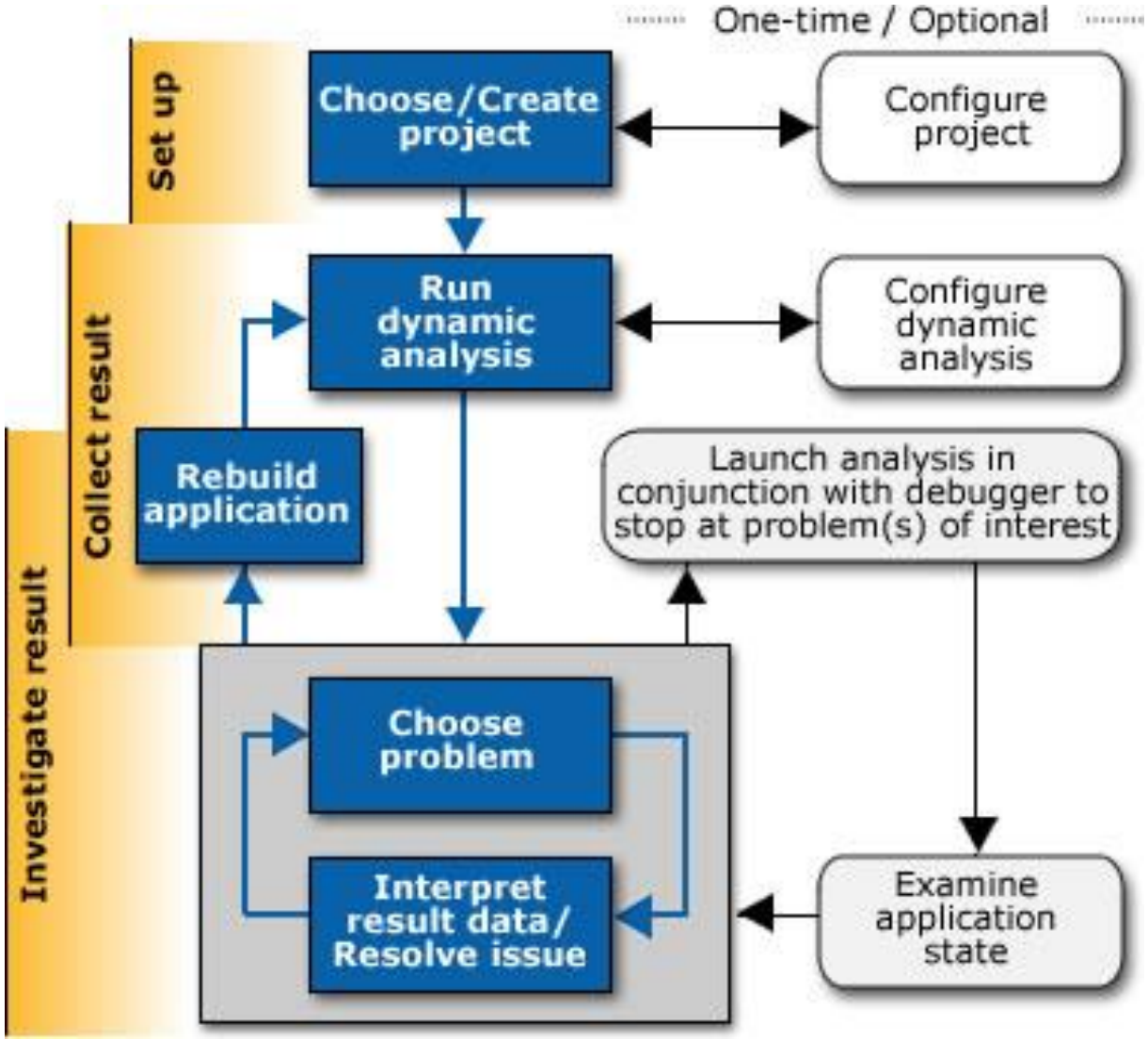


# Standalone GUI: Windows* and Linux*

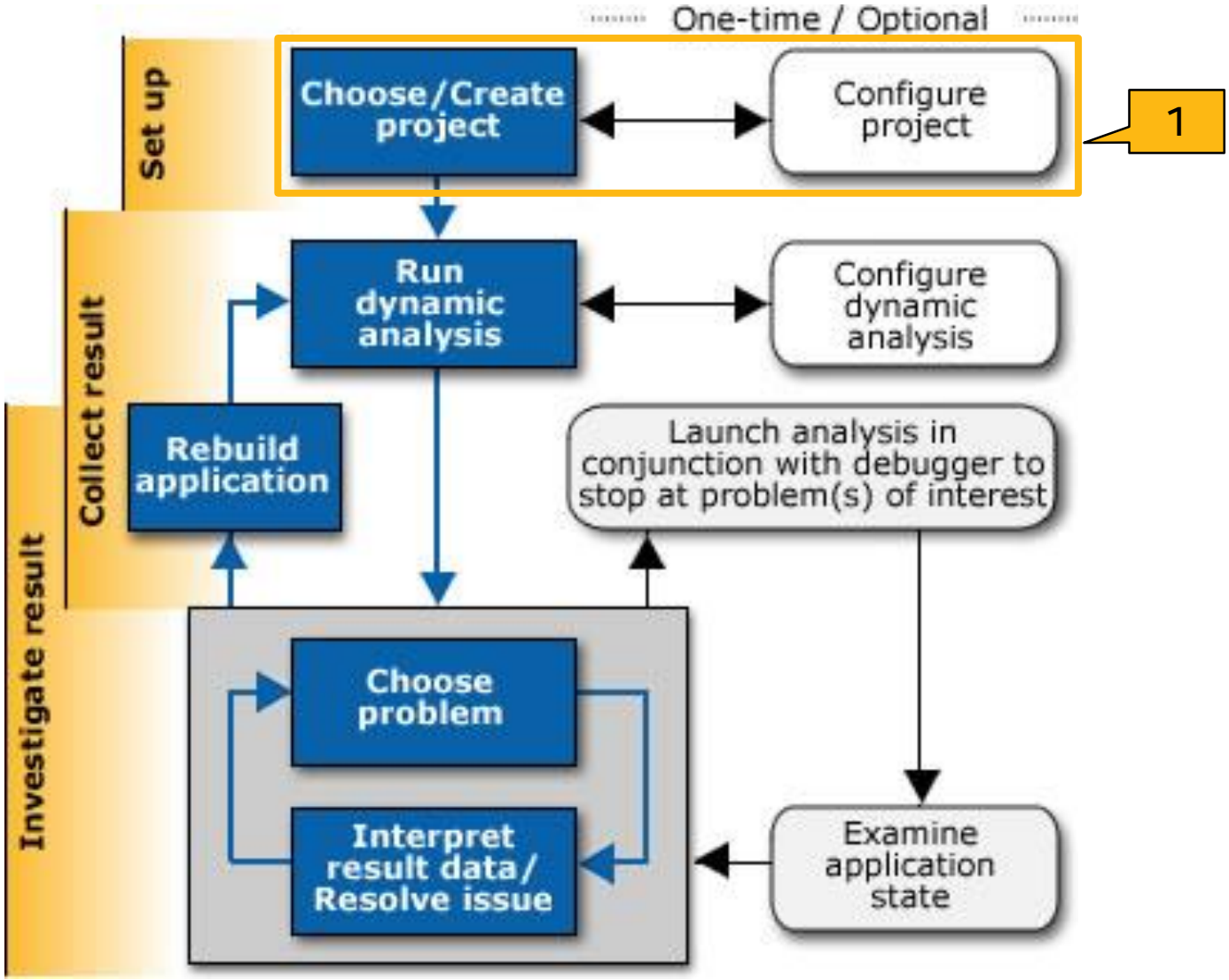


# Analysis workflow

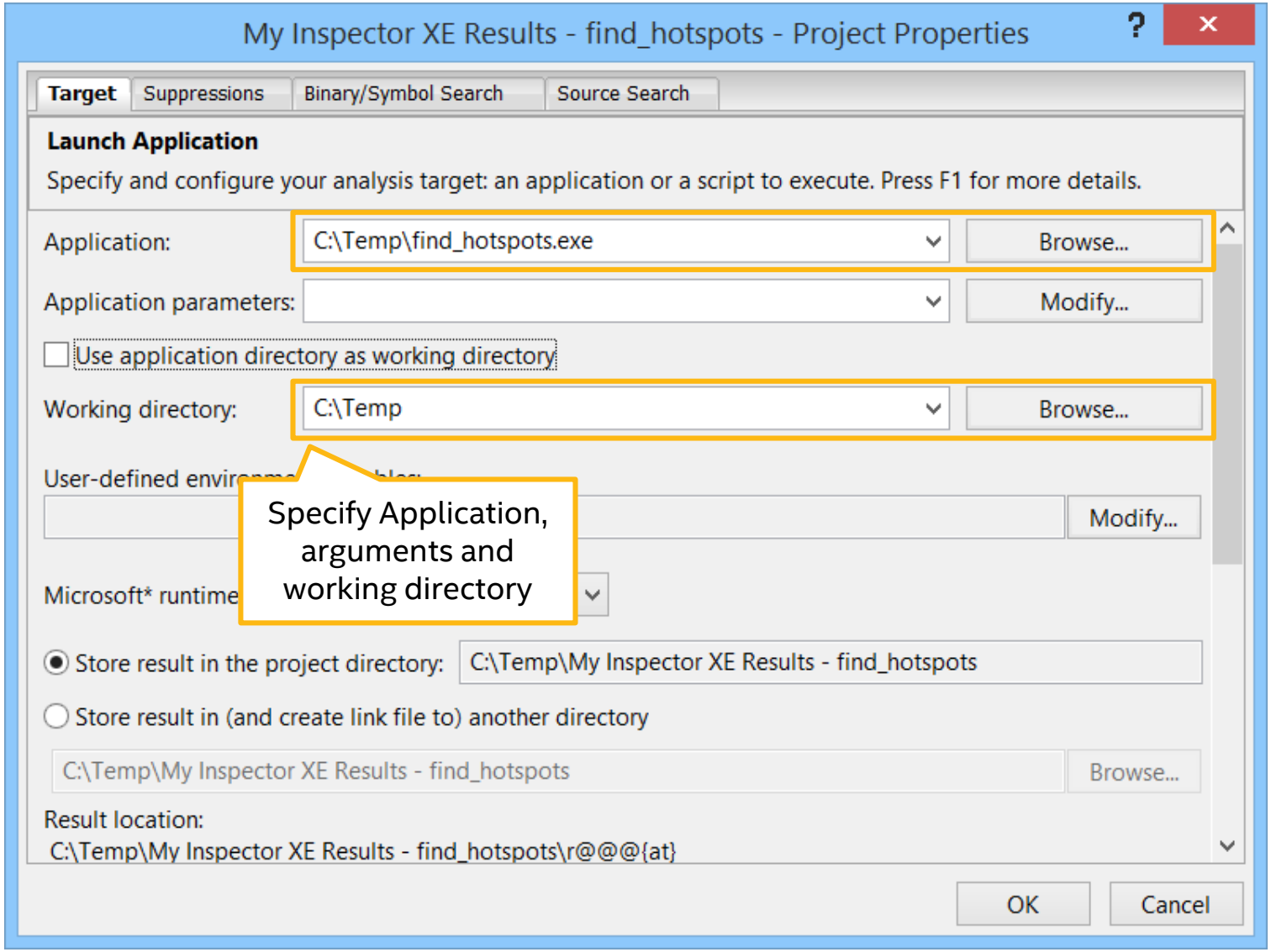
# Workflow: Dynamic Analysis



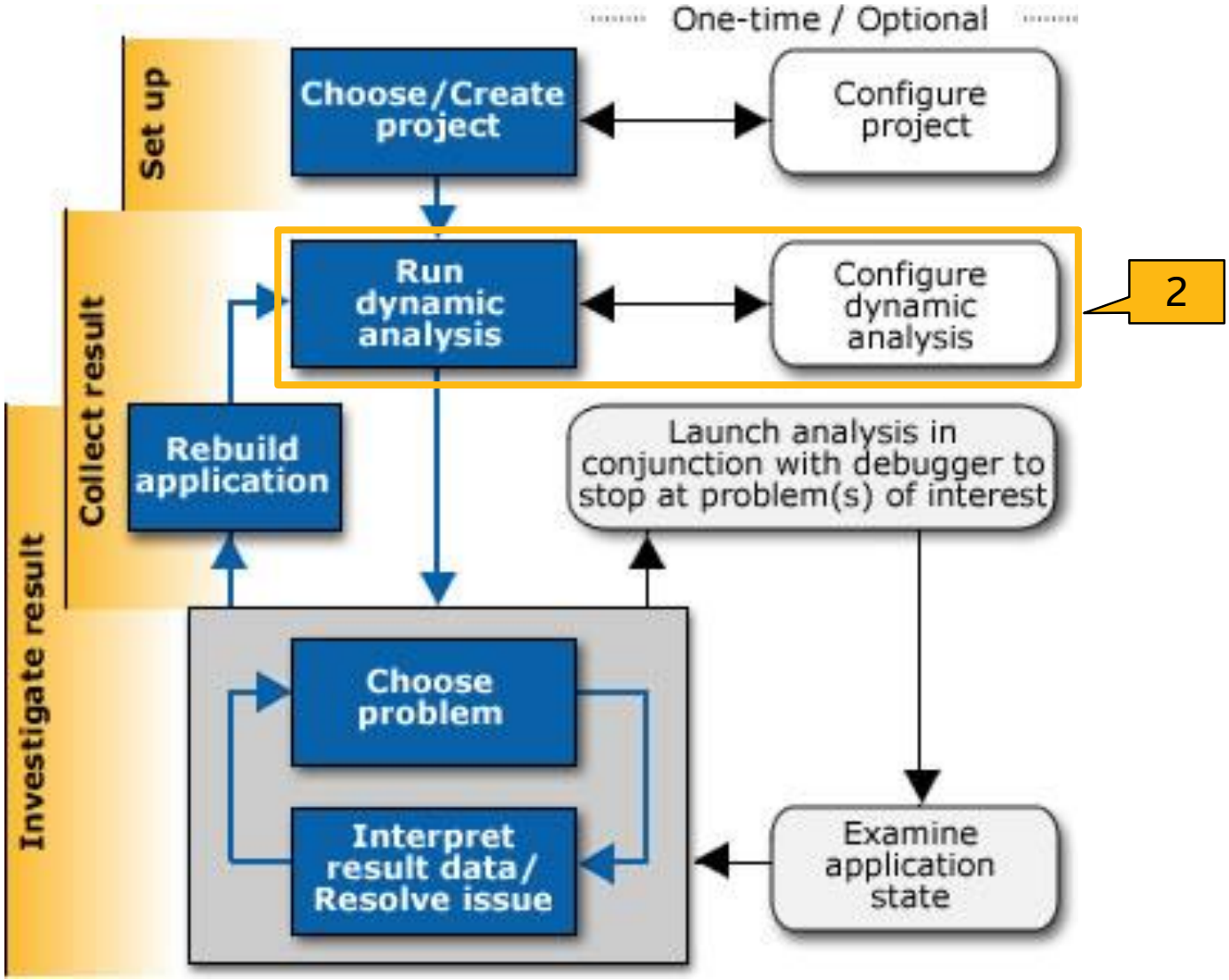
# Workflow: Dynamic Analysis



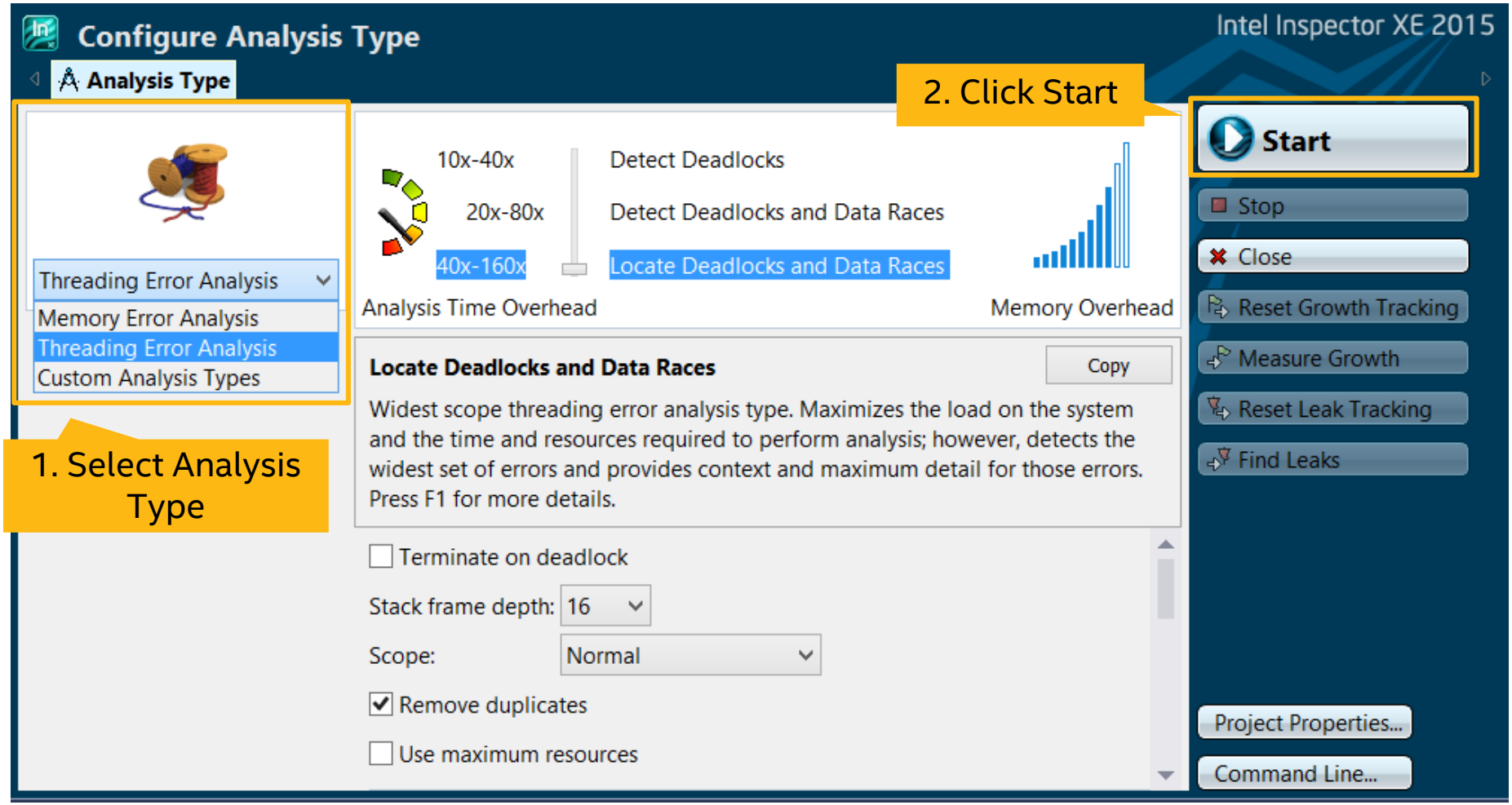
# Workflow: setup project



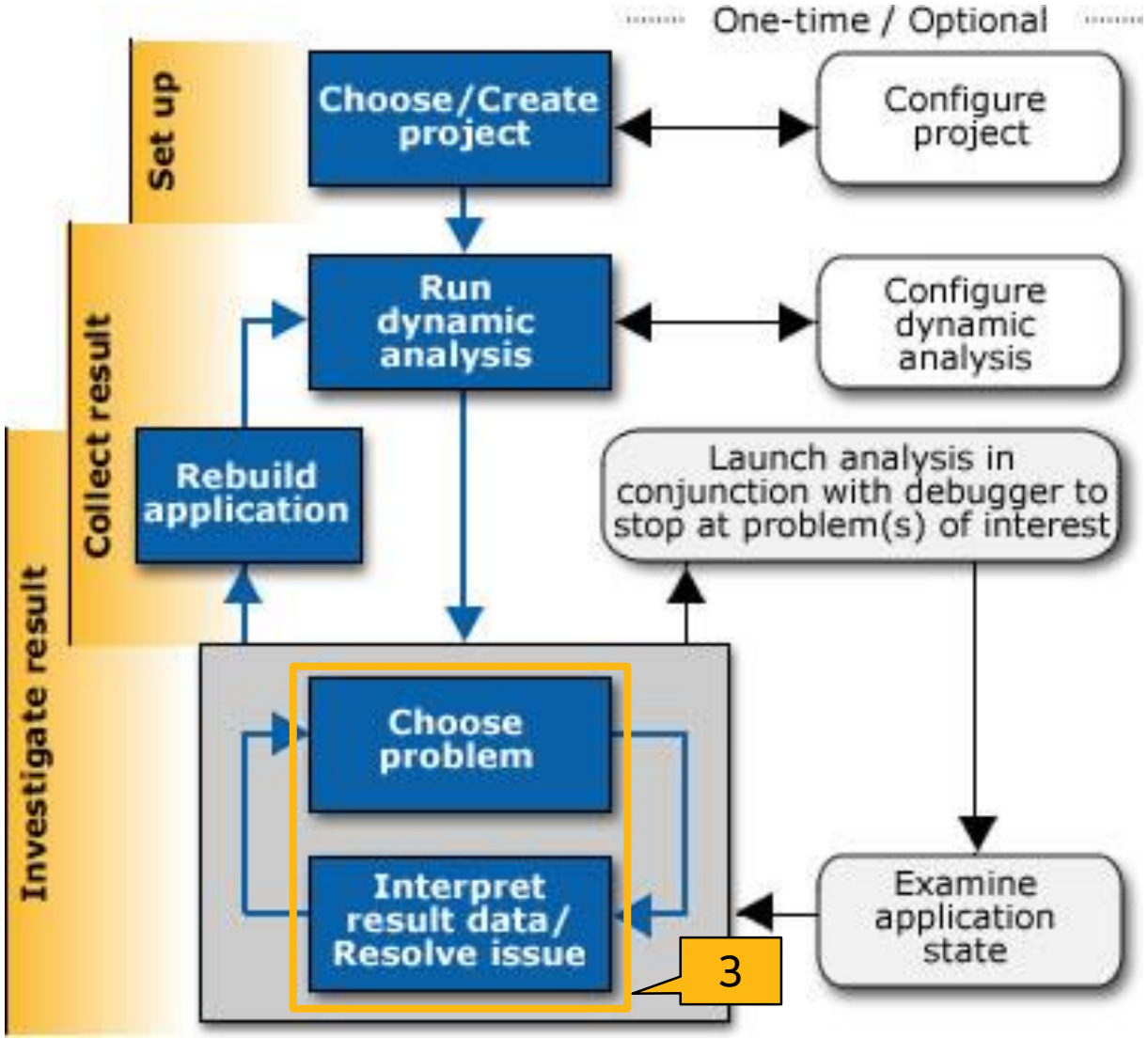
# Workflow: Dynamic Analysis



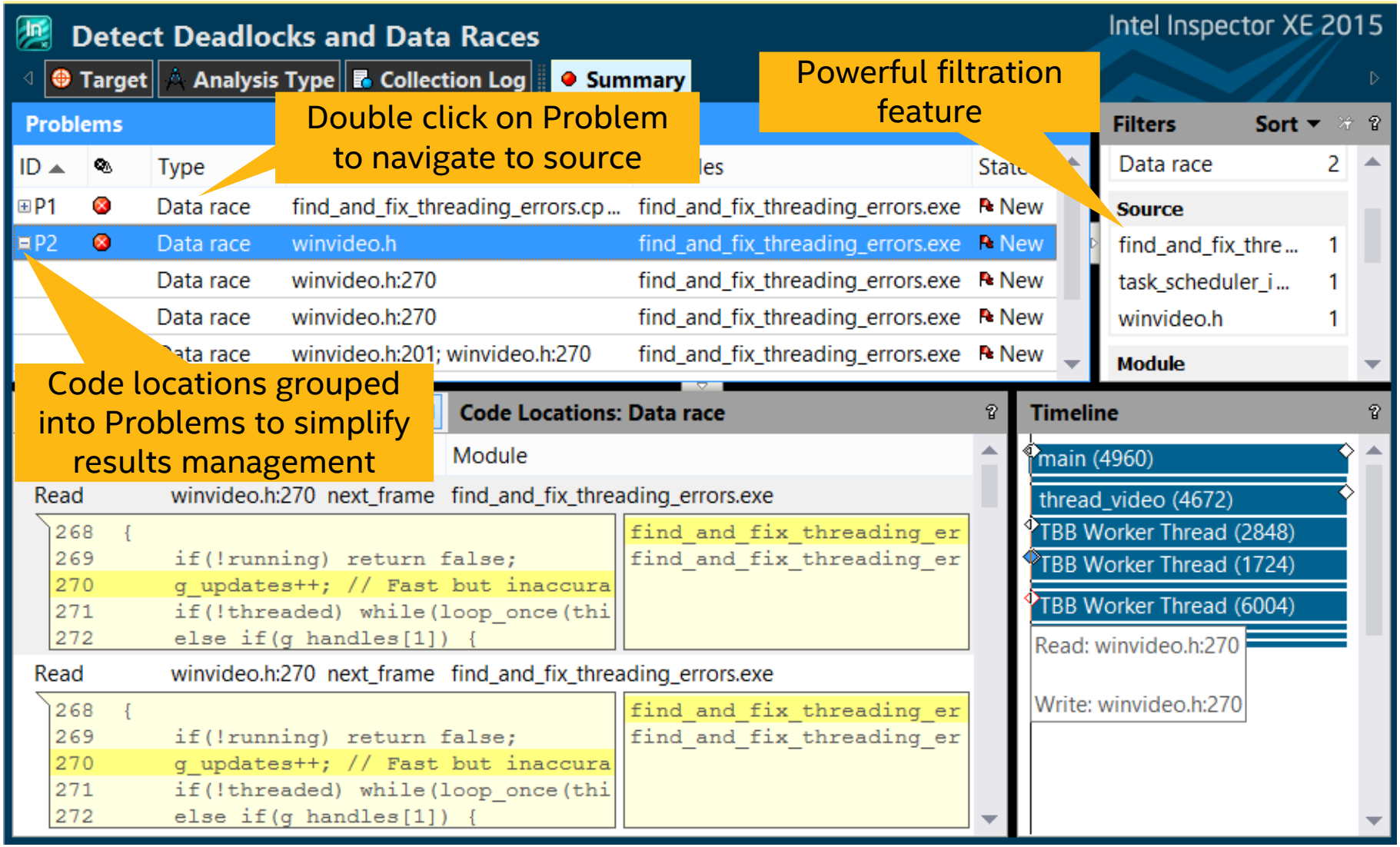
# Workflow: select analysis and start



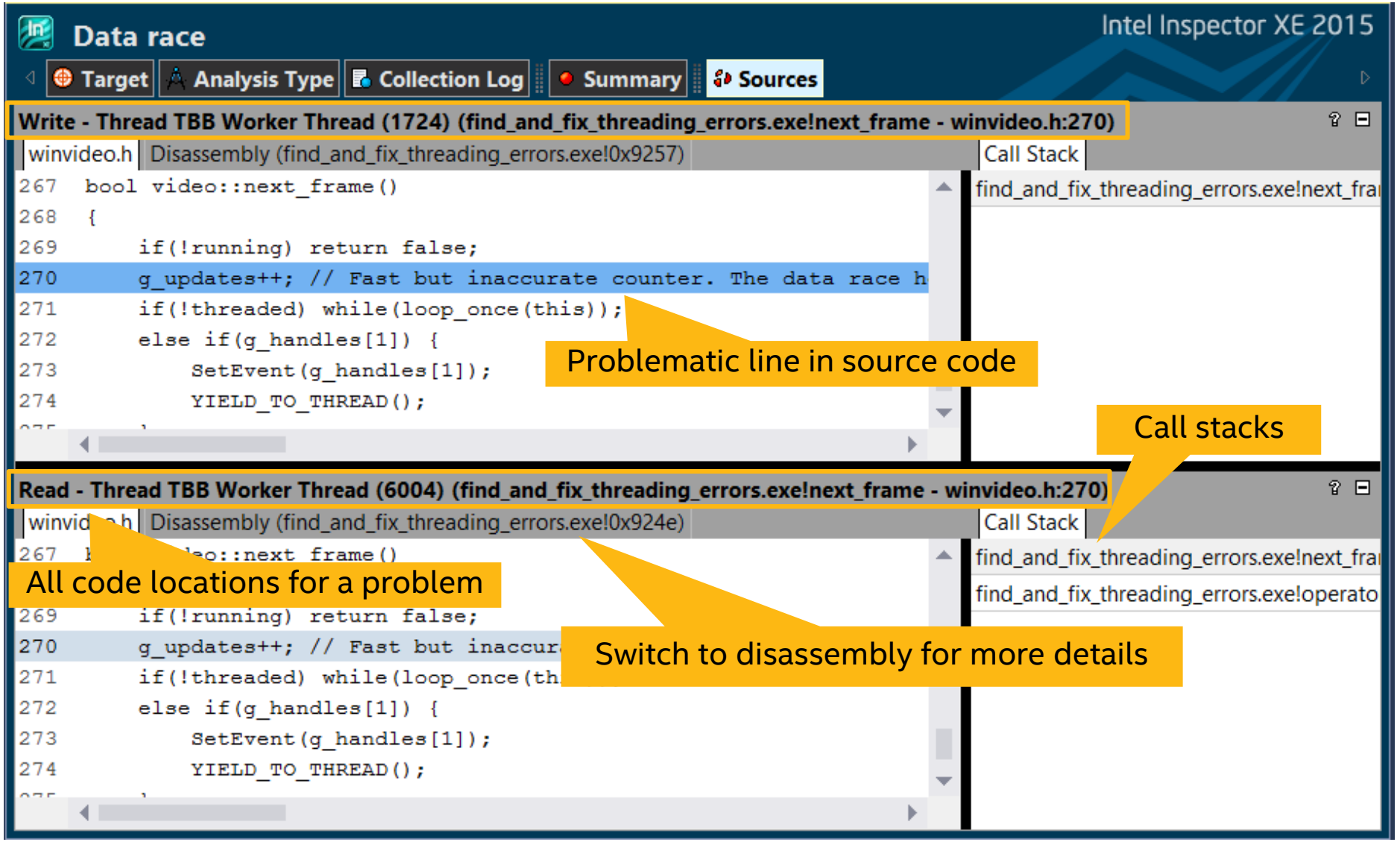
# Workflow: Dynamic Analysis



# Workflow: manage results



# Workflow: navigate to sources



# Memory problem analysis

# Memory problem Analysis

## Analyzed as software runs

- Data (workload) -driven execution
- Program can be single or multi-threaded
- Diagnostics reported incrementally as they occur

## Includes monitoring of:

- Memory allocation and allocating functions
- Memory deallocation and deallocating functions
- Memory leak reporting
- Inconsistent memory API usage

## Analysis scope

- Native code only: C, C++, Fortran
- Code path must be executed to be analyzed
- Workload size affects ability to detect a problem

# Memory problems

## Memory leak

- a block of memory is allocated
- never deallocated
- not reachable (there is no pointer available to deallocate the block)
- Severity level = (**Error**)

```
// Memory leak

char *pStr = (char*) malloc(512);
return;
```

## Memory not deallocated

- a block of memory is allocated
- never deallocated
- still reachable at application exit (there is a pointer available to deallocate the block).
- Severity level = (**Warning**)

```
// Memory not deallocated

static char *pStr = malloc(512);
return;
```

## Memory growth

- a block of memory is allocated
- not deallocated, within a specific time segment during application execution.
- Severity level = (**Warning**)

```
// Memory growth

// Start measuring growth
static char *pStr = malloc(512);
// Stop measuring growth
```

# Memory problems

## Uninitialized memory access

- Read of an uninitialized memory location

```
// Uninitialized Memory Access

void func()
{
   int a;
   int b = a * 4;
}
```

## Invalid Memory Access

- Read or write instruction references memory that is logically or physically invalid

```
// Invalid Memory Access

char *pStr = (char*) malloc(20);
free(pStr);
strcpy(pStr, "my string");
```

## Kernel Resource Leak

- Kernel object handle is created but never closed

```
// Kernel Resource Leak

HANDLE hThread = CreateThread(0,
     8192, work0, NULL, 0, NULL);
return;
```

## GDI Resource Leak

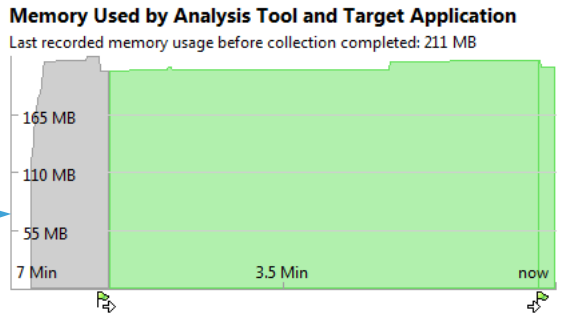
- GDI object is created but never deleted

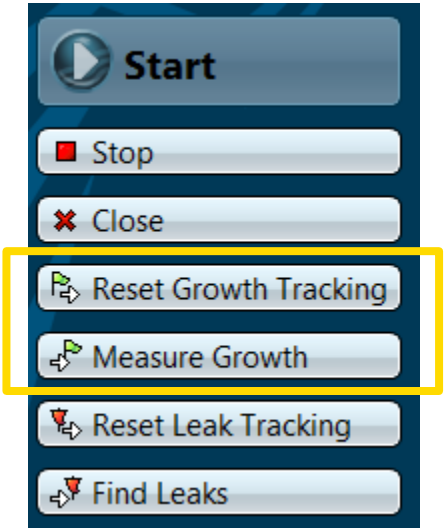
```
// GDI Resource Leak

HPEN pen = CreatePen(0, 0, 0);
return;
```

# Incrementally Diagnose Memory Growth

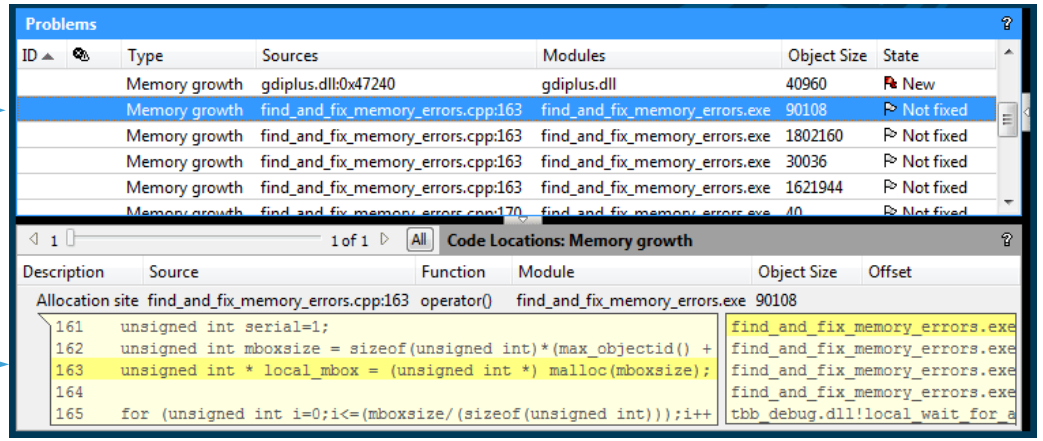
As your app is running…

Memory usage graph plots memory growth




Select a cause of memory growth




See the code snippet & call stack

# Threading problem analysis

# Threading problem Analysis

## Analyzed as software runs

- Data (workload) -driven execution
- Program needs to be multi-threaded
- Diagnostics reported incrementally as they occur

## Includes monitoring of:

- Thread and Sync APIs used
- Thread execution order
  - Scheduler impacts results
- Memory accesses between threads

## Analysis scope

- Native code: C, C++, Fortran
- Managed or mixed code: C# (.NET 2.0 to 3.5, .NET 4.0 with limitations)
- Code path must be executed to be analyzed
- Workload size doesn't affect ability to detect a problem

# Data race

```
CRITICAL_SECTION cs;           // Preparation
int *p = malloc(sizeof(int)); // Allocation Site
*p = 0;
InitializeCriticalSection(&cs);
```

## Write -> Write Data Race

Thread #1

```
*p = 1; // First Write
```

Thread #2

```
EnterCriticalSection(&cs);
*p = 2; // Second Write
LeaveCriticalSection(&cs);
```

## Read -> Write Data Race

Thread #1

```
int x;
x = *p; // Read
```

Thread #2

```
EnterCriticalSection(&cs);
*p = 2; // Write
LeaveCriticalSection(&cs);
```

# Deadlock

```
CRITICAL_SECTION cs1;
CRITICAL_SECTION cs2;
int x = 0;
int y = 0;
InitializeCriticalSection(&cs1); // Allocation Site (cs1)
InitializeCriticalSection(&cs2); // Allocation Site (cs2)
```

Thread #1

```
EnterCriticalSection(&cs1);
x++;
     EnterCriticalSection(&cs2);
     y++;
     LeaveCriticalSection(&cs2);
LeaveCriticalSection(&cs1);
```

Thread #2

```
EnterCriticalSection(&cs2);
y++;
     EnterCriticalSection(&cs1);
     x++;
     LeaveCriticalSection(&cs1);
LeaveCriticalSection(&cs2);
```

## Deadlock

1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #2

## Lock Hierarchy Violation

1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #1
3. EnterCriticalSection(&cs2); in thread #2
4. EnterCriticalSection(&cs1); in thread #2

# Cross-thread Stack Access

```
// A pointer visible for two threads
int *p;
CreateThread(..., thread #1, ...);
CreateThread(..., thread #2, ...);
```

Thread #1

```
// Allocated on Thread #1's stack
int q[1024];
p = q;
q[0] = 1;
```

Thread #2

```
// Thread #1's stack accessed
*p = 2;
```

# Integration with debugger

# Debugger integration

## Break into debugger

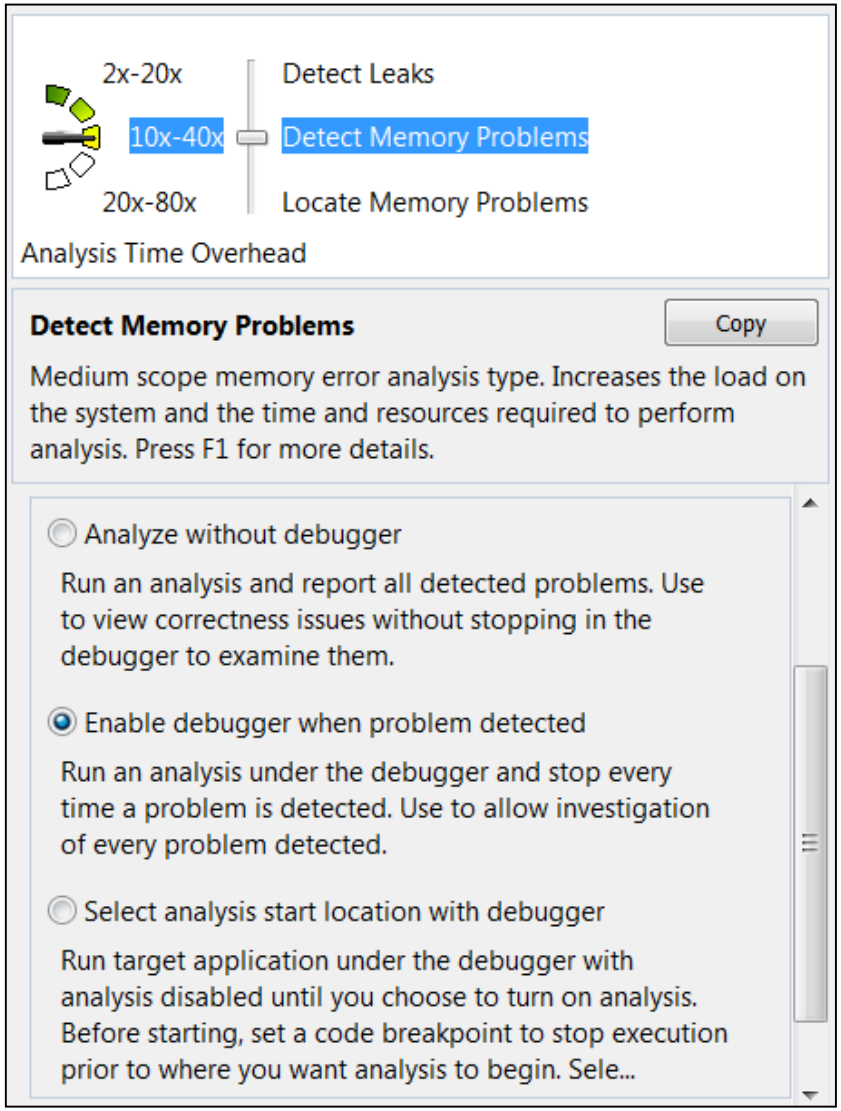
- Analysis can stop when it detects a problem
- User is put into a standard debugging session

## Windows*

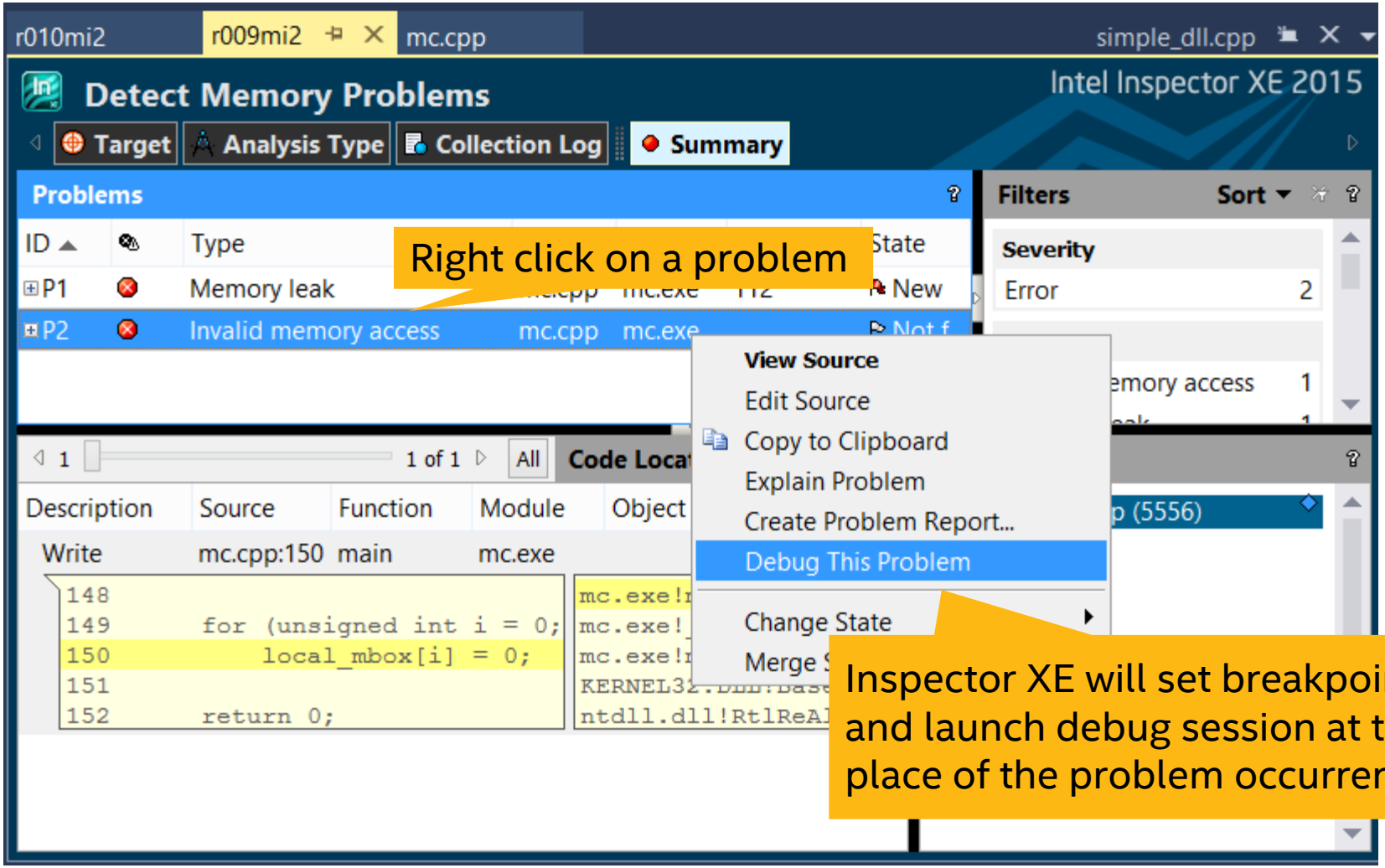
- Microsoft* Visual Studio Debugger (vs2012 – vs2015)

## Linux*

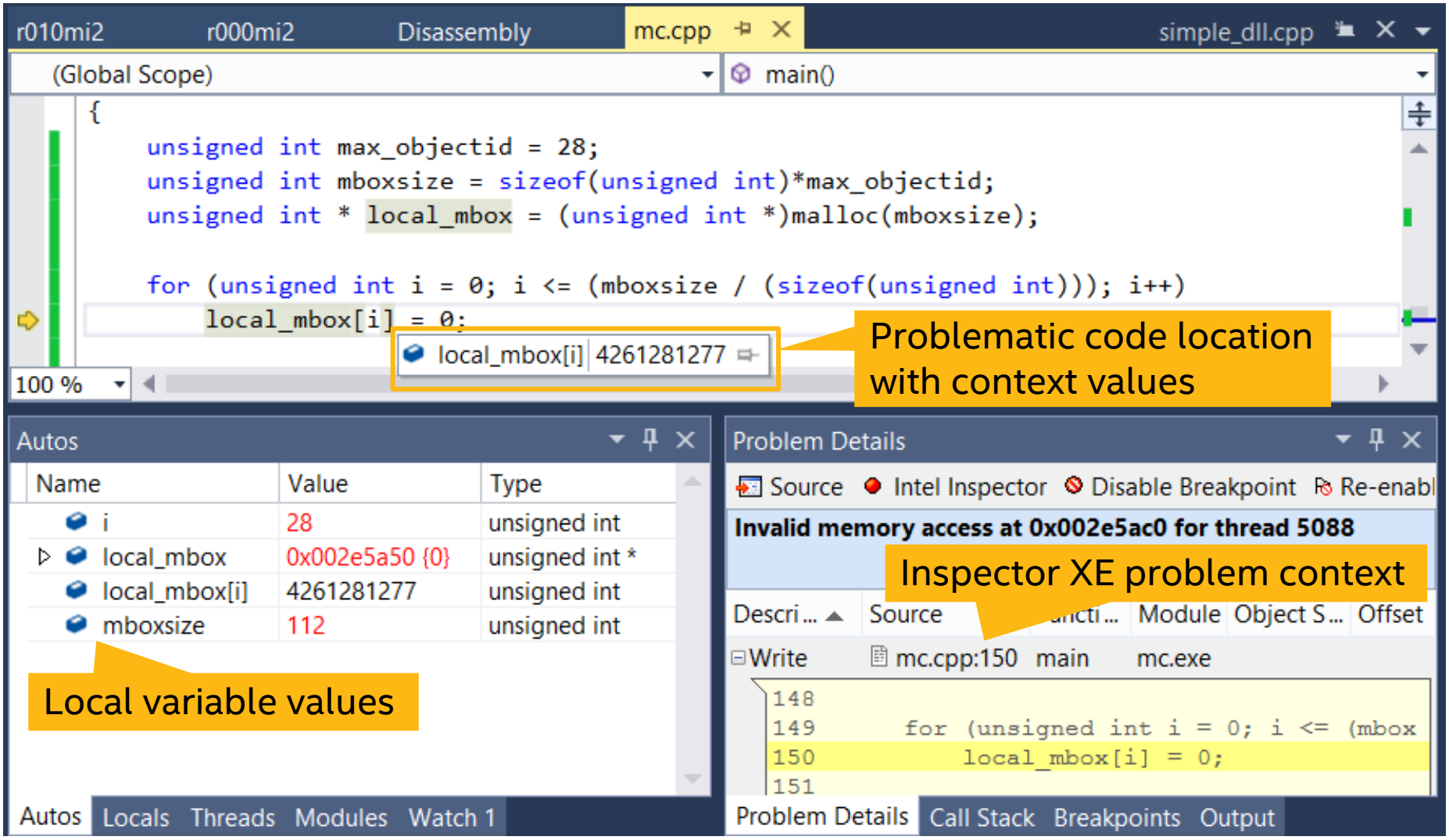
- gdb



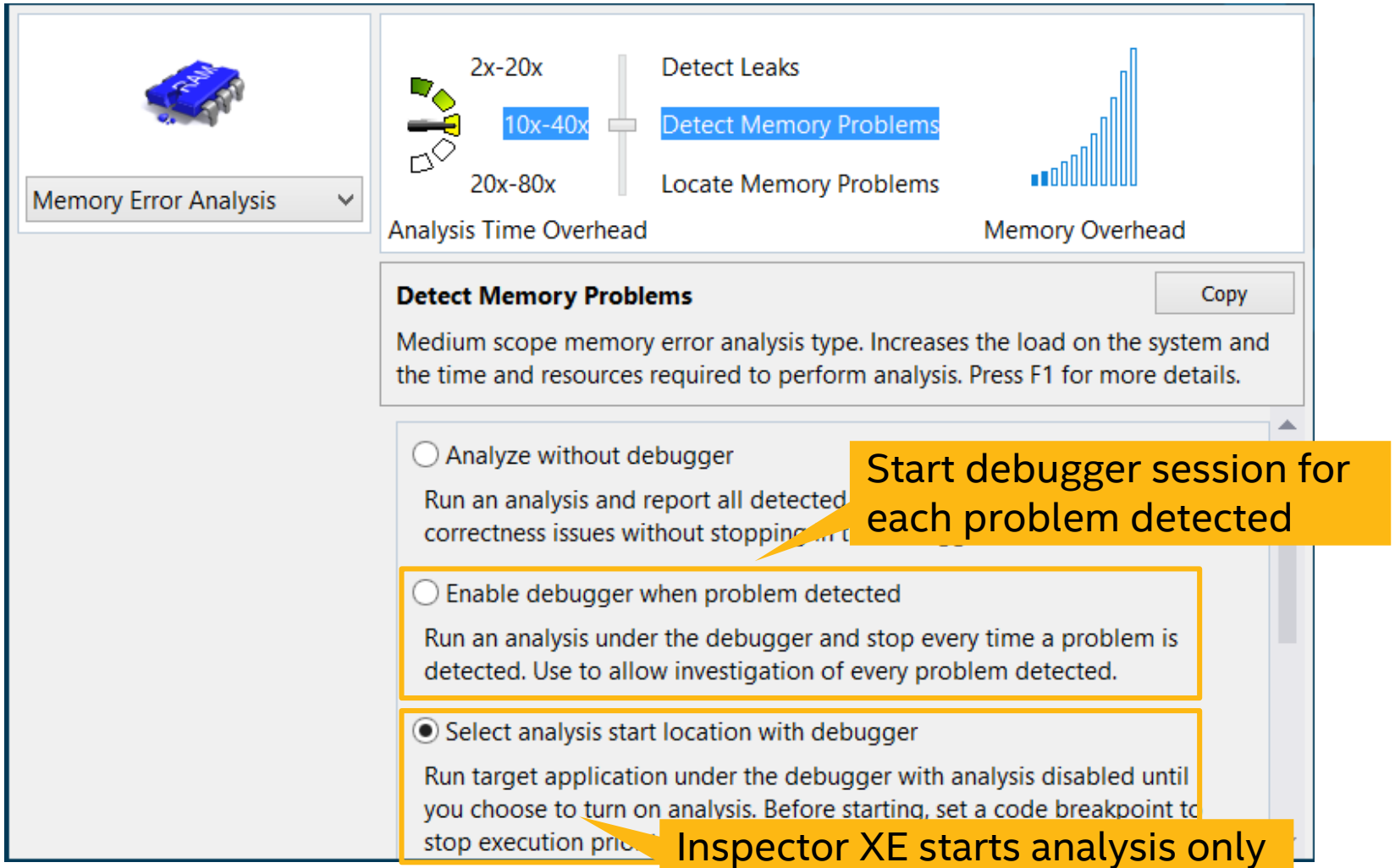
# Debug this problem



# Debug this problem



# Debugger options



# Regression testing and user API

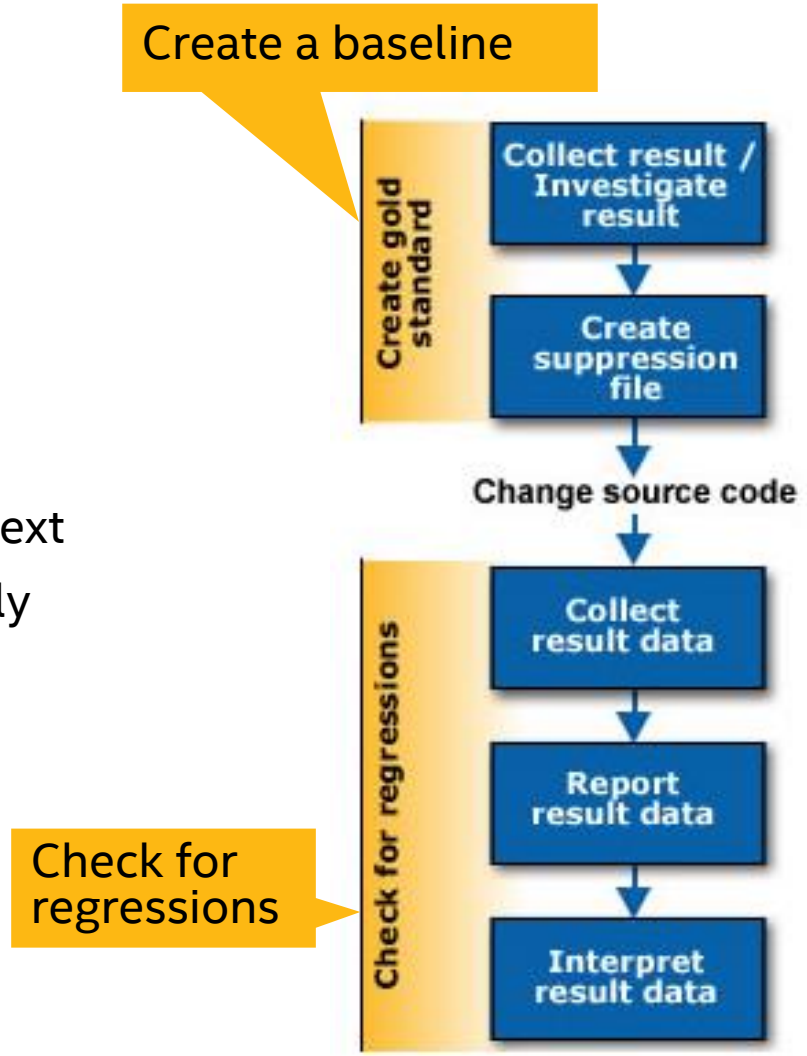
# Automate Regression Analysis

## Data collection from script

- Command line interface (CLI) for running analysis
- Child process analysis

## Reporting CLI

- Exporting results (pack and send)
- Text reports: XML, CSV and plain text
- Detect new problems automatically



# Automate Regression Analysis

## Command Line Interface

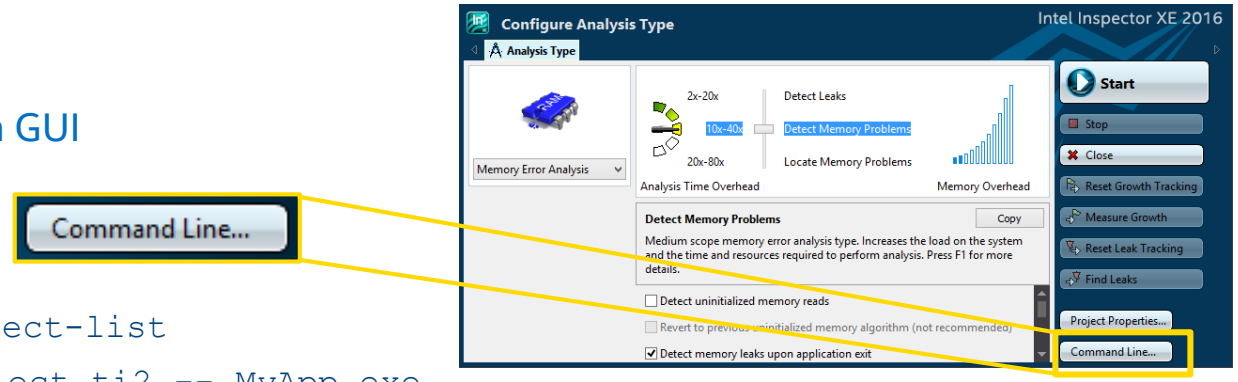
inspxe-cl is the command line:

- **Windows:** `C:\Program Files\Intel\Inspector XE \bin[32|64]\inspxe-cl.exe`
- **Linux:** `/opt/intel/inspector_xe/bin[32|64]/inspxe-cl`

Help:

```
inspxe-cl -help
```

Set up command line with GUI




Command examples:

1. `inspxe-cl -collect-list`
2. `inspxe-cl -collect ti2 -- MyApp.exe`
3. `inspxe-cl -report problems`

# Reporting: regression status

```
inspxe-cl -report status –r r002mi1

9 problem(s) found
    2 Investigated
    7 Not investigated
Breakdown by state:
    2 Confirmed
    4 Not fixed
    2 Regression
    1 New
```

# Intel Inspector XE: User APIs

## Enable you to

- Control collection, limit analysis scope
- Specify non-standard synchronization primitives
- Specify custom memory allocation primitives

## To use user APIs:

- Include `ittnotify.h`, located at `<install_dir>/include`
- Insert `__itt_*` notifications in your code
- Link to the `libittnotify.lib` file located at `<install_dir>/<lib32|lib64>`
- Available for C/C++ and Fortran

# Custom memory allocation

```
#include <ittnotify.h>

__itt_heap_function my_allocator;
__itt_heap_function my_reallocator;
__itt_heap_function my_freer;

void* my_malloc(size_t s)
{
    void* p;

    __itt_heap_allocate_begin (my_allocator, s, 0);
    p = user_defined_malloc (s);
    __itt_heap_allocate_end (my_allocator, &p, s, 0);

    return p;
}
... // Do similar markup for custom “realloc” and “free” operations

// Call this init routine before any calls to user defined allocators
void init_itt_calls()
{
    my_allocator = __itt_heap_function_create("my_malloc", "mydomain");
    my_reallocator = __itt_heap_function_create("my_realloc", "mydomain");
    my_freer = __itt_heap_function_create("my_free", "mydomain");
}
```

# Collection control APIs

| API | Description |
|---|---|
| `void __itt_suppress_push( unsigned int etype)` | Stop analyzing for errors on the current thread |
| `void __itt_suppress_pop ( void)` | Resume analysis |
| `void __itt_suppress_mark_range ( __itt_suppress_mode_t mode, unsigned int etype, void * address, size_t size);` | Suppress or unsuppress error detection for the specific memory range (object). |
| `void __itt_suppress_clear_range ( __itt_suppress_mode_t mode, unsigned int etype, void * address, size_t size);` | Clear the marked memory range |

# User-Defined Synchronization APIs

| API | Description |
|---|---|
| `void __itt_sync_acquired ( void *addr)` | Notify Intel Inspector that synchronization object is acquired by current thread |
| `void __itt_sync_releasing ( void *addr)` | Notify that the code is about to release the specified synchronization object |
| `void __itt_sync_destroy ( void *addr)` | Tell the Intel Inspector that the synchronization object will not be used again, so the Intel Inspector can dispose of bookkeeping information associated with this object. |

# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

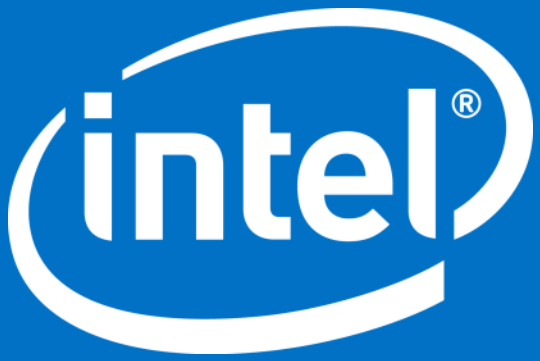
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.



**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel®