POINTER CHECKER: Easily Catch Out-of-Bounds Memory Accesses

by Kittur Ganesh, *Technical Consulting Engineer*

This article introduces a powerful new feature called Pointer Checker, which precisely and easily isolates elusive bugs in programs. Found in the Intel® C++ Composer XE 2013 product, its integration into the compiler adds powerful functionality in a way that slides seamlessly into build systems. Clever implementation and powerful error reporting provide precise information about latent program defects. We are excited that during beta testing of this new feature, customers reported that this tool found numerous defects.

Although C/C++ pointers have well-defined semantics, many applications could still make out-of-bounds memory accesses which can go undetected, risking data corruption and increasing vulnerability to malicious attacks. The Pointer Checker provides full checking of all memory accesses through pointers. A pointer-checked enabled application will therefore catch out-of-bounds memory accesses before memory corruption occurs.

With the advent of multicore processors, there is a need to program for data and thread parallelism where data is frequently created, stored, shared, and accessed in memory through pointers. The C and C++ languages define good semantics for memory access through pointers, but they also permit the use of these pointers without any restrictions. This provides no built-in protection against accessing or writing most user data in memory. This means you can perform any number of arbitrary operations on the pointers—resulting in severe unforeseen errors in the program whose effects often appear random due to unintentional modification of data—causing out-of-bounds (OOB) memory accesses which may often go undetected.

Although pointers have well-defined lower and upper bounds, languages (and therefore the compilers) typically don't enforce bounds checking due to performance and speed concerns. This paves way for potential buffer overflows and overruns in various parts of the application code—causing data corruption, erratic program behavior, breach of system security, etc.—and is the basis for many software vulnerabilities to malicious attacks.

The launch of Intel® Parallel Studio XE 2013 brings a key new feature: the Pointer Checker, which performs bounds checking—providing full checking of all memory accesses through pointers—and identifies any out-of-bounds access in Pointer Checker-enabled code. This article presents a comprehensive overview and usage model of Pointer Checker, enabling you to quickly get started using this key debugging feature on your critical applications.

Overview

The Pointer Checker is a key feature of Intel Parallel Studio XE 2013. The main functionality of Pointer Checker is to find buffer overflows or overruns occurring in applications developed in high-level C and C++ languages on Windows* or Linux* operating systems. A buffer overflow or a buffer overrun is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety. For example, consider an array as the buffer as shown in the short code snippet in **Figure 1**.

```
char *buf = (char *)malloc(5);
for (int i=0; i<=5;i++) {
 buf[i] = 'A' + i;
}
```

Figure 1

A buffer overflow occurs when you try to put more items in the array than what the array can hold. It occurs generally from writing or a store operation. On the other hand, a buffer overrun occurs when you are iterating over the buffer and keep reading past the end of the array. It generally occurs from reading or a load operation. Additionally, simple coding errors are often very hard to locate and rectify. For example, pointers are invariably masked by casting to a void pointer and then recasting to other pointers, making it very difficult to identify the cause of errors in the application. As mentioned earlier, since a pointer has a well-defined lower and upper bound, Pointer Checker performs bounds checking for all memory accesses through pointers—ensuring that a pointer is within bounds before its use for either a read or a write operation.

The Pointer Checker feature can be enabled via *compile time switches*. When you build your application with the Pointer Checker-enabled option, it will identify and report out all out-of-bounds memory accesses occurring in the application, including subscripted array accesses. In addition, the Pointer Checker can also detect *dangling pointers*, meaning pointers that point to memory that has been freed. When you build your application with the dangling pointer detection-enabled option, using a dangling pointer in an indirect access will also cause the Pointer Checker to report out an out-of-bounds error. Another useful feature that Pointer Checker offers is to check bounds for *arrays without dimensions*, which is especially important since applications are integrated with many different modules developed by different developers who often extern shared data.

The Pointer Checker feature is implemented mostly in a runtime library which is automatically linked in by the Intel® compiler. It also offers an easy-to-use, user application programming interface (API) to allow control over what happens when an out-of-bounds violation is detected. You can be assured that Pointer Checker does not change the data structure layout or application binary interfaces (ABI), thus allowing your application to contain Pointer Checker-enabled code, as well as code that is not enabled. This is a key benefit of Pointer Checker: you can *incrementally enable* and test a small number of critical files, gradually enabling the rest of the legacy code in your application.

To ensure pointers have proper bounds set, there are a few well defined *intrinsics* that you can use to retrieve, kill, or make new pointer bounds. This is especially handy when working with a mix of enabled and non-enabled code or, for example, when using your own custom allocation functions that may need resetting of pointer bounds. Also, the Pointer Checker contains a Runtime Library Functions (RTL) wrapper library used for checking of C runtime library functions, so proper bounds are maintained for pointers manipulating memory in those functions referenced in the application.

One obvious concern is the *runtime performance overhead* incurred when application is enabled with Pointer Checker since it entails bounds checking of all pointers in memory. For this reason, the Pointer Checker is seen as a key debugging feature designed for use during application testing and debugging prior to releasing the product.

"The main functionality of Pointer Checker is to find buffer overflows or overruns occurring in applications developed in high-level C and C++ languages on Windows* or Linux* operating systems."

High-Level Design

The high-level design philosophy for Pointer Checker is "*bounds follow the pointer*." The compiler creates bounds when a pointer is created via the "&" operator or array reference, copies bounds when a pointer is copied, stores bounds when a pointer is stored in memory, loads bounds when a pointer is loaded from memory, and passes bounds with pointer arguments and function returns. In addition, the compiler generates checks when a pointer is used for indirect memory references.

The runtime library wrapper functions contained in the wrapper library create bounds when memory is allocated, copies bounds for memory copies, and checks bounds for pointer parameters (e.g., `strcpy()`). Also, it has to be noted that casting does not change the bounds of a pointer, thus maintaining consistency across function calls and MACRO expansions.

Usage Model

The Pointer Checker usage model is simple and easy to understand and is made up of compiler options for enabling Pointer Checker, intrinsics for retrieving or manipulating pointer bounds, reporting API functions and enumerations for controlling how out-of-bounds are reported and, finally, RTL wrapper functions to ensure that the pointer's usage within the RTL routines is within bounds. A header file "`chkp.h`" contains the definitions of intrinsics and reporting API for inclusion in the Pointer Checker-enabled application. **Table 1** provides a summary of features covered in more detail in subsequent sections.

Model	DESCRIPTION	
Header File	Defines intrinsics and reporting functions (chkp.h)	
Compiler Options	-check-pointers (On Linux*) /Qcheck-pointers (On Windows*)	Enables pointer checker and adds associated libraries
	-check-pointers-dangling (/Q-check-pointers-dangling)	Enables checking for dangling pointer references
	-[no-]check-pointers-undimensioned (/Qcheck-pointers-undimensioned[-])	The default is to check the undimensioned arrays. You use the switch to DISABLE checking of undimensioned arrays for nonstandard code.
Intrinsics	void * __chkp_lower_bound(void **)	Returns the lower bound associated with the pointer
	void * __chkp_upper_bound(void **)	Returns the upper bound associated with the pointer
	void * __chkp_kill_bounds(void *p)	Removes the bounds information and pointer in argument can access all memory.
	void * __chkp_make_bounds(void *p, size_t size)	Creates a new pointer with bounds specified by arguments. The bounds are attached to the returned pointer. The pointer that is passed as argument is not affected.
Reporting API (Function/ Enumeration)	void __chkp_report_control(__chkp_report_option_t option, __chkp_callback_t callback)	Determines how errors are reported
	__chkp_report_option_t {Key Enumerations: __CHKP_REPORT_LOG, __CHKP_REPORT_TRACE_LOG, __CHKP_REPORT_CALLBACK, __CHKP_REPORT_BPT, __CHKP_REPORT_TERM}	Controls how out-of-bounds errors are reported. Enumerations are defined in the header file "chkp.h"
RTL Functions	Provides checking on C runtime library functions that manipulate memory through pointers	

Table 1

Using Pointer Checker

As stated earlier, Pointer Checker can be enabled with a single compiler switch to check bounds of pointers for write or read/write operations. In conjunction, you can check for arrays with and without dimensions, check for dangling pointers, check runtime library functions, check custom memory allocators, and so on. In this section, real-world sample scenarios are used to illustrate the various ways with which you can use Pointer Checker to ensure memory safety and catch out-of-bounds memory accesses, if any, in your application.

Checking Bounds

With a single compiler switch, you can enable Pointer Checker to check bounds of pointers on read/write operations. (**Figure 2**)

By default, Pointer Checker is disabled. If you compile with the write option, Pointer Checker is enabled and checks for bounds of pointers for all *write* operations. Specifying the "rw" option ensures bounds checks for all *read*, as well as *write* operations. Consider the real-world example in **Figure 3** that was part of a large enterprise application code base.

Compile and execute **without** Pointer Checker-enabled switch using the Intel® compiler as follows:

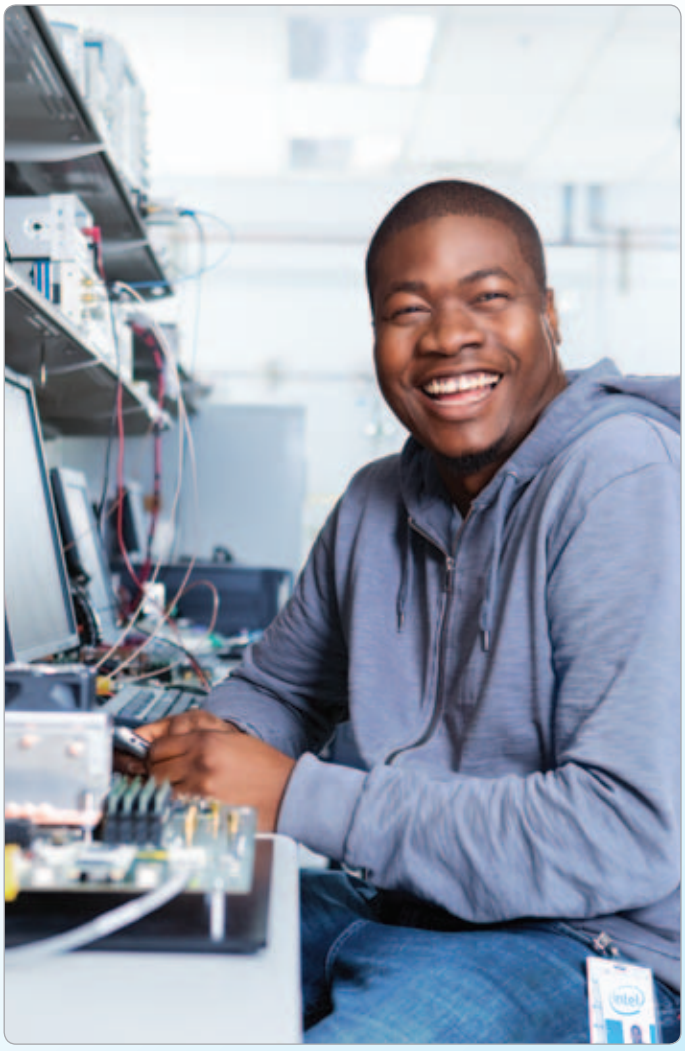
```
% icc main.c;./a.out
sizeof another_chptr is 3
sizeof my_chptr is 8
after memset = @@@@@@@@
```

Notice that eight characters are output instead of the intended three characters. This is because the developer made a common, subtle programming error in passing the size of the pointer "`my_chptr`" instead of the string length of the object pointed to by `my_chptr` pointer. Let's find out if Pointer Checker catches this out-of-bounds (OOB) error following the steps outlined below.

Compile and execute with Pointer Checker enabled for write operations. (**Figure 4**)

In the example code, the enumeration `option __CHKP_REPORT_TRACE_LOG` is used in the reporting `library function __chkp_report_control()` which tells Pointer Checker to report out OOB error with call trace-back information from where OOB occurred. Also, the option `–g (/Zi on Windows*)` and the linker option –rdynamic is used for better trace-back output.

"Although C/C++ pointers have well-defined semantics, many applications could still make out-of-bounds memory accesses which can go undetected, risking data corruption and increasing vulnerability to malicious attacks."



```
-check-pointers=[none | write | rw] (On Linux* OS)
/Qcheck-pointers:[none | write | rw] (On Windows* OS)
```

Figure 2

```
1 #include <stdio.h>
2 #include <chkp.h>
3
4 int main () {
5 #ifdef __MYCP_REPORT
6 __chkp_report_control(__CHKP_REPORT_TRACE_LOG, 0);
7 #endif
8 char *my_chptr[] = “abc”;
9 char *another_chptr;
10 another_chptr = (char *) malloc (strlen(( char *)my_chptr));
11 printf (“sizeof another_chptr is %d\n”, strlen(( char *)my_chptr));
12 printf (“sizeof my_chptr is %d\n”, sizeof(my_chptr));
13 memset (another_chptr, ‘@’, sizeof(my_chptr)); /* OOB */
14 printf (“after memset = %s\n”, another_chptr);
15 return 0;
16 }
```

Figure 3

```
% icc main.c –D__MYCP_REPORT -check-pointers=write -rdynamic -g; ./a.out
sizeof another_chptr is 3
sizeof my_chptr is 8
CHKP: Bounds check error
lb: 0x232e010
ub: 0x232e012
addr: 0x232e017
end: 0x232e017
size: 1
Traceback:
./a.out(__chkp_memset+0x6b) [0x40448b]
in file unknown line 0
./a.out(main+0x334) [0x402f48] in file main.c line 13
/lib64/libc.so.6(__libc_start_main+0xfd) [0x3b7d41ec5d]
in file unknown line 0
./a.out() [0x402b59]
```

Figure 4

```
1 int ppmatch(char *s, char *t) {
2 char c;
3 while(isupper(*s) || isupper(*t)) {
4 if (*s != *t) return FALSE;
5 s++;
6 t++;
7 }
8 while (*s != '\0') {
9 if (*s != '#') {
10 if (*t == '\0') c = '*'; else c = *t;
11 if (*s != c) return FALSE;
12 }
13 s++;
14 t++;
15 } return TRUE;
16 }
```

Figure 5

```
1 #include <stdio.h>
2 #include <malloc.h>
3 #include <chkp.h>
4 int main () {
5 #ifdef __MYCP_REPORT
6__chkp_report_control(__CHKP_REPORT_TRACE_OG, 0);
7 #endif
8 char *buf = malloc(4);
9 int i;
10 for (i=0; i<=4; i++) {
11 printf(" %c",buf[i]); /* OOB */
12 }
13 for (i=0; i<=4; i++) {
14 buf[i] = 'A' + i; /* OOB */
15 printf(" %c",buf[i]); } printf ("\n"); return 0;
16 }
```

Figure 6

You'll notice that Pointer Checker indeed catches an OOB error and outputs the trace-back from where the fault occurred, which is at the address: `./a.out(main+0x334)[ 0x4032b8]`. The source line where OOB occurs is at line number 13 in the file main.c for mapped address `0x4032b8`. This clearly is the line containing the `memset()` call where the size of the pointer "`my_chptr`" was used instead of the intended string length of `my_chptr` pointer. The trace-back output also gives bounds information, such as the lower bound (lb), upper bound (ub), and the address information of the pointer where the OOB fault occurred.

Figure 5 shows yet another tricky real-world code snippet (parser code in one of SPEC 2000 benchmarks) that ends up accessing beyond the zero terminator of the string. If the string passed in pointer s is longer than in pointer t, t can get incremented past the end of the string in the execution of the benchmark and gets an OOB error from Pointer Checker. **Figure 6** shows another classic code example containing a buffer overrun in the first for loop (line# 11) during a read operation where the iterator goes past the upper bound of 4 bytes allocated for pointer buf, and a buffer overflow during a write operation in the next for loop (line# 14). Note that you'll need to compile with "rw" option for check-pointers so read/write operations are checked by Pointer Checker for OOB errors.

```
-check-pointers-dangling=[none | heap | stack | all] (Linux* OS)
/Qcheck-pointers-dangling:[none | heap | stack | all] (Windows* OS)
```

Figure 7

```
1 #include <stdio.h>
2 #include <chkp.h>
3 char *foo()
4 {
5 char fr[]="test"; /* func: char *GetUserInput()*/
6 char *to = malloc(10);
7 strcpy(to, fr);
8 free(to);
9 return to;
10 }
11 int main()
12 {
13 #ifdef REPORT
14 15 __chkp_report_control(__CHKP_REPORT_TRACE_LOG, 0);
15 #endif
16 char *sp = foo();
17 printf("first ch=%s\n",*sp); /* OOB */
18 }
```

Figure 8

```
-[no-]check-pointers-undimensioned (Linux* OS)
/Qcheck-pointers-undimensioned[-] (Windows* OS)
```

Figure 9

"Pointer Checker-enabled code and non-enabled code can coexist. Security benefits from catching software vulnerabilities prior to product release far outweigh the runtime performance overhead, which is the big trade-off."

```
%cat main.c
1 #include <stdio.h>
2 #include <chkp.h>
3 extern int A[];
4
5 int main () {
6 #ifdef REPORT
7__chkp_report_control(__CHKP_REPORT_TRACE_LOG, 0);
8 #endif
9 A[3] = 1;
10 A[5] = 2; /* OOB */
11 return 0;
12 }
%cat arr1.c
int A [5]
```

Figure 10

```
Perl -4 code snippets:

Building the application results in:
   doio.obj : error LNK2019: unresolved external symbol
   __cp_array_end__sys_errlist referenced in function _nextargv
   perl.obj : error LNK2001: unresolved external symbol
   __cp_array_end__sys_errlist
   stab.obj : error LNK2001: unresolved external symbol
   __cp_array_end__sys_errlist
   perl4.exe : fatal error LNK1120: 1 unresolved externals

1) sys_errlist is declared in perl.h and undimensioned array sys_errlist is
   then referenced in the definition of strerror:

   perl.h: #define strerror(e) ((e) < 0 || (e) >= sys_nerr ? "(unknown)" : sys_
   errlist[e])

2) strerror is called in doio.c,perl.c, and stab,c hence the 3 unresolved
   symbol messages
```

Figure 11

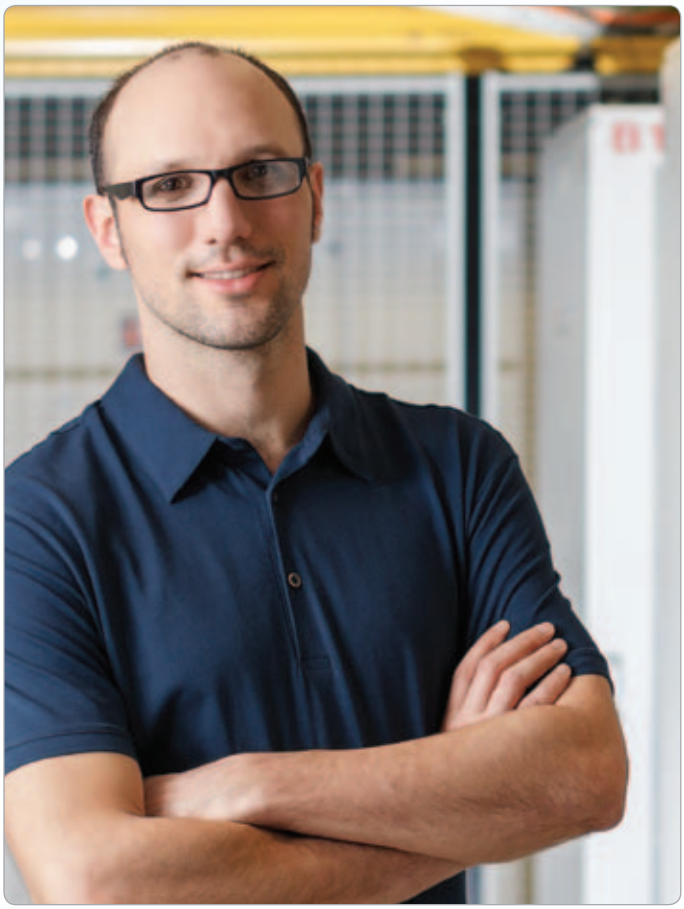
Checking for Dangling Pointers

Pointer Checker can identify and report dangling pointers on the stack or heap, and can be enabled by a single compiler switch in **Figure 7**.

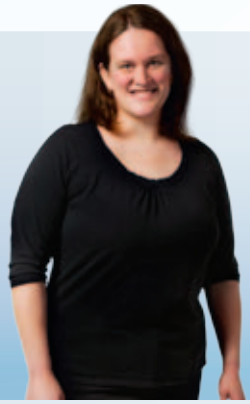
Pointer Checker uses a wrapper function for the runtime routine free() and the C++ delete operator, and sets the lower bound of all the dangling pointers in heap or stack to 2 and upper bounds to `0;` so any pointer with bounds values as such can be considered a dangling pointer in your Pointer Checker-enabled application. **Figure 8** shows a careless mistake by the developer: freeing the "to" pointer before return in function foo() and, when enabled for dangling pointers, using the option –check-pointers=heap, Pointer Checker will report an OOB error at line number 17.

Checking Arrays (With or Without Dimensions)

Pointer Checker, when enabled for read / write, checks for bounds of arrays defined with dimensions by default. For checking of bounds of arrays *without* dimensions, you can use the compile switch as in **Figure 9. Figure 10** shows another real-world example where an array A defined in file arr1.c is extern-ed in another file arr.c and then written to it beyond the allocated size of 5 elements. When the code is Pointer Checker enabled for write operation, Pointer Checker will report an OOB at line number 10 in the code.



BLOG highlights



Minimize frustration and maximize tuning effort with Amdahl's Law

BY SHANNON CEPEDA

I recently had a question from a customer who had introduced a successful optimization to a hot function in his application, but did not see as much improvement in the overall application as he expected. This is a fairly common occurrence in the iterative process of performance tuning. Usually it happens for one of two reasons.

1. Introducing an improvement in one area resulted in inefficiencies somewhere else. This is par for the course with performance tuning, and part of the reason why the process is **iterative**. It can be hard to anticipate whether a code change you are making in one function will decrease performance somewhere else down the road, and so landing in this situation from time to time is unavoidable. Although you may not be able to always prevent it, using good documentation practices and a tool like Intel® VTune™ Amplifier XE to quantify performance changes can help you see when it is happening.

SEE THE REST OF SHANNON'S BLOG:

Visit **Go-Parallel.com**

Browse other blogs exploring a range of related subjects at Go Parallel: Translating Multicore Power into Application Performance.

```
%nm libchkpwrap.a | egrep 'T __chkp_' |grep strcpy
00000000000002a0 T __chkp_strcpy
```

Figure 12

```
Intrinsics:
void * __chkp_kill_bounds(void *p)
> Kills the descriptor associated with the
pointer making all memory accessible via the
returned pointer.
Void * __chkp_make_bounds(void *p, size_t size)
> Make bounds for a pointer. The lower
bounds is pointer, and the upper bound is pointer +
(size – 1).
void * __chkp_lower_bound(void **) /
void * __chkp_upper_bound(void **)
> Retrieves the lower / upper bound
associated with a pointer
```

Figure 13

```
Example: custom memory allocator
void *myalloc(size_t size) {
// Code allocating the large chunk of memory
// into smaller chunks.
// Add bounds information to the pointer
p = __chkp_make_bounds(p, size);
return p;
}
```

Figure 14

```
void *myalloc(size_t old_size) {
// code
…..
// code creating a pointer by using a RTL function from a non-enabled module
(void *) p = my_realloc(p, old_size + 100);
p = __chkp_kill_bounds(p);
p = (void*)__chkp_make_bounds(p, old_size + 100);
return p;
}
```

Figure 16

Checking for un-dimensioned arrays is especially important, as arrays defined in a module by one developer are referenced as un-dimensioned in other modules developed by other developers, and vice-versa. **Figure 11** shows code snippets of interest from an older version of Perl 4 application. In it, the un-dimensioned array "`sys_errlist`" is declared in perl.h and referenced in strerror macro definition, which then is used in several files like doio.c, perl.c, stab.c, etc.

Building the Perl 4 application with Pointer Checker enabled for read / write operations results in unresolved external symbols of `__cp_array_end___sys_errlist` messages. Note that, Pointer Checker adds the prefix `__cp_array_end` to the un-dimensioned array `sys_errlist` symbol for upper bound notation as it cannot find the dimensioned array definition during compilation. Also, the dimensioned array definition happens to be defined in a non-pointer-checker-enabled library. The solution for such a scenario is to turn off un-dimensioned checks by using `–check-pointers-undimensioned (Linux*)` or `/Qcheck-pointers-undimensioned- (Windows*)` and complete the build.

A similar approach can be used in scenarios where, for example, an array "`a`" is defined in one module with a dimension (say 100), again redefined in another module with a dimension (say 200), and referenced as an un-dimensioned array "`a[]`" in a different module, which C-language does allow, although it's not a standard approach. Compiling such an application enabled for checks for un-dimensioned arrays would result in "`multiple definition of 'a'`" and "`multiple definition of `__cp_array_end_a`'" messages, as Pointer Checker would have added the prefix `__cp_array_end` for upper bound of symbol "`a`." In such a scenario, the solution is to turn off un-dimensioned arrays checks for relevant files when enabling with Pointer Checker for read or write operations.

Checking Runtime Library Functions

There are many C runtime library functions that manipulate memory through pointers and may need to be encapsulated or replaced so returned pointers have proper descriptors, and so usage of pointers within the RTL routine are checked correctly for any bounds violations. Pointer Checker has a wrapper library called libchkpwrap (Linux) and libchkpwrap.lib (Windows), which provides equivalent C runtime library wrapper functions that either replace the runtime library function or wrap with appropriate pointer-checking mechanisms. The wrapper library is automatically linked-in when you compile with the Intel® compiler with Pointer Checker enabled. All equivalent wrapper runtime library functions will have a prefix "`__chkp`" attached to the name of the library. For example, equivalent `strcpy()` wrapper library function (Linux) can be found as shown in **Figure 12**.

You can also write your own wrappers for runtime library functions. You will then have to manipulate the pointer bounds, for which you would use one or more of the Pointer Checker intrinsics.

Using Intrinsics

The Pointer Checker intrinsics are ideal for writing your own wrappers for runtime library functions, when working with Pointer Checker-enabled and non-enabled modules, or when you are creating custom memory allocators and need bounds manipulation (checking and creating correct bounds), etc. **Figure 13** shows the available intrinsics to kill bounds associated with a pointer, make bounds, or retrieve bounds information. **Figure 14** demonstrates how you can write a custom wrapper for manipulating a large chuck of memory into smaller chucks and ensure bounds are checked. It is important to note that the returned pointer from the __chkp_make_bounds() call has the new bounds.

Figure 15 shows a sample program demonstrating a custom allocator wrapper using Pointer Checker intrinsics.

Working with Enabled and Non-Enabled Modules

An enabled module is a module compiled with the Pointer Checker-enabled switch, while a non-enabled module is compiled with Pointer Checker option disabled. If you write a pointer to memory or return a pointer from a non-enabled module, the pointer may get incorrect bounds information. If you use this pointer with the incorrect bounds information in an enabled module, the pointer checker will report an incorrect out-of-bounds error because the bounds do not correspond

"A pointer checked-enabled application will catch out-of-bounds memory accesses before memory corruption occurs."

to the pointer. Pointer Checker mitigates this for nearly all cases by checking the pointer against a copy of the pointer stored with the bounds information. If there is a mismatch, the bounds are set to allow access to any memory. An OOB error can still be reported, for example, when memory is extended by a `realloc()` in a non-enabled module, but bounds aren't reset to new bounds. To prevent such incorrect OOB errors the solution is to write wrapper functions in non-enabled module that kills or sets the bounds correctly for any pointer returned or written by the functions.

For instance, consider a case where a pointer is created by a RTL function `my_realloc()` from a non-enabled module as shown in **Figure 16**. If the memory allocator simply extends the memory allocated to pointer p and then returns the same pointer, an enabled module could use this pointer that has old bounds information. The Pointer Checker then reports an out-of-bounds error because it doesn't know about the extension created by the realloc() function. The solution is to remove the existing bounds information from the pointer p and make new bounds using __chkp_kill_bounds() and __chkp_make_bounds() intrinsic functions.

"The Pointer Checker intrinsics are ideal for writing your own wrappers for runtime library functions, when working with Pointer Checker-enabled and non-enabled modules, or when you are creating custom memory allocators and need bounds manipulation (checking and creating correct bounds), etc."

Identifying and reporting Out-of-Bounds Errors

The Pointer Checker provides a library function "`__chkp_report_control()`" to control reporting of OOB errors found in the Pointer Checker-enabled application. The function takes a reporting control enumeration value for the first argument and the second parameter is NULL except when the enumeration is the call back function value for which a user-defined call back function is specified for the second argument. Table 1 shows the enumerations, which are also defined in the header file "`chkp.h`" located under the compiler include directory. Most commonly used enumerations are: `__CHKP_REPORT_LOG` which logs the OOB errors and continues until all errors are reported, and `__CHKP_REPORT_TRACE_LOG` which reports stack trace-back including instruction addresses from the OOB fault in the call chain. For example, **Figure 4** shows the trace-back output for the enumeration `__CHKP_REPORT_TRACE_LOG` which is similar to back-trace output in the GNU or Intel® debugger.

Guidelines

When using Pointer Checker, use debug configuration (with –g option on Linux and /Zi on Windows) for testing and debugging so symbols are seen for better trace-back functionality. Use the –rdynamic linker option when compiling on Linux, so function names are output in the trace-back. And, compile with no optimization to avoid optimizing away memory accesses and also improve source code correlation. First catch OOB errors for write operations and then for read, since writes are more critical and can cause severe software vulnerabilities. Use `__CHKP_REPORT_LOG` in conjunction with `__CHKP_REPORT_TRACE_LOG` report to analyze loop specific issues, as fixing one OOB error will fix all the loop-related OOB errors right away.

Release your application with the Pointer Checker option disabled, as application size and execution time increases with Pointer Checker-enabled applications. Runtime cost is high, about 2X-5X the execution time (based on some open source application runs), and code size increases from 20 percent to 100 percent or more depending on the application.

Summary

Pointer Checker is a key feature of the Intel® Parallel Studio XE 2013 suite. Pointer Checker is a debug tool designed for application debugging and testing. Pointer Checker-enabled code and non-enabled code can coexist. Security benefits from catching software vulnerabilities prior to product release far outweigh the runtime performance overhead, which is the big trade-off. **Get started with Pointer Checker and catch any out-of-bounds memory accesses before any memory corruption occurs.**

Appendix

A Sample Custom Allocator Wrapper for Pointer Checker

The example below demonstrates a custom allocator wrapper using Pointer Checker intrinsics:

```
//===========================================================
// SAMPLE SOURCE CODE - SUBJECT TO THE TERMS OF SAMPLE CODE LICENSE AGREEMENT,
// http://software.intel.com/en-us/articles/intel-sam-
ple-source-code-license-agreement/
//
// Copyright 2012 Intel Corporation
//
// THIS FILE IS PROVIDED "AS IS" WITH NO WARRATIES, EXPRESS OR IMPLIED, INCLUDING BUT
// NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
// PURPOSE, NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS.
// ===========================================================
//
// Toy memory allocator to demonstrate how to write custom allocator
// wrappers for check-pointers
//
// Allocates a large pool using malloc, then parcels out small pieces
// from it. By creating bounds for these parcelled out pieces their
// bounds will be automatically checked by check-pointers as well.
//
// Compile with -check-pointer=rw
// Run as:
// a.out len string len string len string
//
// Each pair of arguments will first allocate len bytes from the larger
// memory pool, and create bounds for it. It will then copy the next
// argument into this parcel as a string. For example:
//
// a.out 2 a 3 ab 3 abc
//
// The third pair will incur a bounds violation, whereas without
// check-pointers the program would be perfectly fine.

#include <stdio.h>
#include <stdlib.h>

#if defined(__INTEL_CHKP_ENABLED)
#include <chkp.h>
#endif

void *pool_base = (void*)0;
size_t pool_size = 0;
```

Figure 15

```
void init_pool(size_t sz)
{
 pool_base = malloc(sz);
 if (!pool_base) {
 fprintf(stderr,
 "Error: unable to allocate pool_base to size %lld\n",
 (long long)sz);
 }
 pool_ptr = pool_base;
 pool_size = sz;
}

void *get_one(size_t sz)
{
 void *p;
 if (sz < pool_size) {
 p = pool_ptr;
 pool_ptr += sz;
 pool_size -= sz;
#if defined(__INTEL_CHKP_ENABLED)
 p = (void*)__chkp_make_bounds(p, sz);
#endif
 return p;
 }
 fprintf(stderr, "Error: memory pool exhausted.\n");
 exit(0);
}

int
main(int argc, char *argv[])
{
 int i;
 init_pool(64);
 for (i = 1; i < argc; i+=2) {
 int sz = atoi(argv[i]);
 char *str = get_one(sz);
 if (i+1 < argc) {
 strcpy(str, argv[i+1]);
 }
 fprintf(stdout, "Copied to %#p \"%s\"\n", str, str);
 }
}
```

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