



Debugging Dynamic Memory Usage Errors Using NonStop Native Inspect

Table of contents

Introduction	3
Intended Audience	3
Typographic Conventions	3
Related Information	4
Prerequisites	4
Limitations	5
Memory-Related Errors	5
Heap Corruption	5
Causes for Heap Corruption	6
Memory Leaks	6
When to Suspect a Memory Leak	6
Types of Memory Leaks	6
Access Errors	7
Using Native Inspect to Debug Memory Problems	7
Memory-Debugging Features of Native Inspect	8
Heap Profiling	8
Example 1: Filtered Heap Reporting for Allocations Exceeding <num> at a Particular Call-Site	10
Leak Profiling	10
Event Monitoring	12
Monitoring Heap Events	12
Monitoring Allocations Greater Than a Specified size	12
Example 2: Monitoring allocations greater than a specified size	13
Monitoring De-allocations to Detect Double-Frees	13
Monitoring Heap Corruption	14
Monitoring String Corruption	14
Example 3: Monitoring heap-corruption caused by erroneous handling of string functions	14
Detecting Out-of-Bounds Writes with the Bounds-Checking Feature	16
Example 4: Bounds-checking to detect out-of-bounds writes	16
Detecting Heap Corruption	17
Example 5 Detecting heap corruption using the info corruption command	17
Settings to Manage Performance Degradation	18
Supported Modes of Memory-debugging in Native Inspect	19
Debugging Programs Launched Under Debugger Control	19
Debugging a Running Process	20
Summary of Memory Debugging Commands	20

Conclusion.....	20
Additional Examples	21
Example 6 Detecting a double-free error	21
Example 7 Detecting de-allocation of memory that has not been initialized	21
Example 8 Detecting de-allocation of unallocated blocks.....	22
Example 9 Detecting memory leaks caused when an application overwrites a pointer that currently addresses a block of memory with another address or data.....	23
Example 10 Detecting memory leaks that are caused when a pointer variable in an application addresses memory that is out of the scope of the application	24
Example 11 Detecting memory leaks when you free a structure or an array that has pointers which are not freed.....	25
FAQ.....	26

Introduction

HP NonStop Debugger (Native Inspect) is an HP-supported implementation of the open source debugger GDB. In addition to the normal debugging functions, it also enables you to debug memory-related errors in a program.

Native Inspect supports memory-debugging (using Run Time Checking (RTC)) of source-level programs written in HP NonStop C and HP NonStop C++ on Itanium®-based systems running the NonStop Guardian or OSS operating systems.

Native Inspect offers the following memory-debugging capabilities:

- Reports memory leaks
- Reports heap allocation profile
- Stops program execution if bad writes occur with string operations such as `strcpy()`, and `memcpy()`
- Stops program execution when freeing unallocated or deallocated blocks
- Stops program execution when freeing a block if bad writes occur outside block boundary
- Stops program execution conditionally based on whether a specified block address is allocated or de-allocated

Intended Audience

This document is intended for C and C++ programmers who use NonStop Native Inspect debugger to detect and debug memory-related errors. The reader of this document must be familiar with the basic commands supported by Native Inspect.

Typographic Conventions

This document uses the following typographical conventions:

:	A colon represents the system prompt for OSS.
>	A right arrow represents a prompt in TACL on Guardian.
Command	A command name or qualified command phrase.
Computer output	Text displayed by the computer.
ENVIRONMENT VARIABLE	The name of an environment variable, for example, <code>PATH</code> .
[ERROR NAME]	The name of an error, usually returned in the <code>errno</code> variable.
Variable	The name of a placeholder in a command, function, or other syntax display that you replace with an actual value.
< >	The contents are optional in syntax. If the contents are a list separated by <code> </code> , you must choose one of the items.
	Separates items in a list of choices.
IMPORTANT	This alert provides essential information to explain a concept or to complete a task.
Note	A note contains additional information to emphasize or supplement important points of the main text.

Related Information

The NonStop C runtime library defines several functions useful in detecting memory-related problems. These functions are not part of the memory leak detection functionality in the debugger. The function `heap_check()` will perform a sanity check over the memory management internal data structures. The function `heap_check_always()` turns on heap checking in the runtime library enabling checking for every memory-related call into the runtime library (for example a call to `malloc()`). This method of doing memory checking is more limited than what is described here. It will not compute any memory leak information nor are there any checks for improper use of calls to string functions. Another drawback of using these functions is the need to recompile and relink the application. Note: on Guardian OS the user may turn on library checking using this unsupported method without the need to recompile:

```
ADD DEFINE =_HEAP_CHECK_ALWAYS_,class map,file on
```

The debugger command, `info corruption`, is roughly equivalent to the `heap_check()` function.

Note:

`heap_check()` may be invoked using a command line call from within the Native inspect debugger. This is simply way to do a very basic check of the heap consistency at a specific execution point.

For more information see the following NonStop documents:

TNS C Library Calls Reference Manual

Guardian Native C Library Calls Reference Manual

Open System Services Library Calls Reference Manual

The Native Inspect Manual is available at the following location:

<http://docs.hp.com/en/528122-008/528122-008.pdf>

Prerequisites

Following are the prerequisites for debugging memory-related problems in Native Inspect:

- NSK Native Inspect debugger (T1237) H06.15 running on NSK H06.15 or later.
- Native Inspect uses the heap debugging library, `zrtcdll`, to enable memory-debugging support. The `zrtcdll` library is a part of the NonStop Native Inspect product. If the library is installed in a directory other than the default `$(SYSTEM.SYS00)` versioned subvolume, you may specify the library location using the `LIB` option. In this text this subvolume will always be referred to as `SYS00`.
- The program must be launched with the user library `zrtcdll`. (A user library is introduced on the command line using the `lib` option. See an example below in the Heap Profiling section.)
- The user must enable heap checking from within the debugger. (See `heap-check on` for more details.)
- The memory-debugging feature is supported only for programs that directly or indirectly call `malloc()`, `realloc`, `calloc()`, or `free()` from the standard C library, `$(SYSTEM.SYS00.ZCREDLL)`.

Limitations

The following are some of the limitations in the use of memory leak detection in Native Inspect.

- The memory debugging feature is not supported for threaded programs as of H06.15.
- The program can not require a user library (a “user library” is a special library specified in the `lib` option). Only one user library is allowed and that is used for the `zrtcdll` library.
- Memory leak reports and so on can not be generated from a process which has been suspended at stop or abend.
- The memory debugging feature can not be used with applications that redefine or override the default system-supplied versions of the standard library routines (under `zcredll`, and `zcrtdll`), such as `abort()`, `strcat()`, and `dlclose()`. If there are doubts about the applications' use of memory functions, one can use the `enoft` utility. `enoft` may be used to determine if the application or the dependent libraries in the application redefine or substitute the standard library routines. For more information on the dependent standard library routines, see the Native Inspect Manual.
- Memory leak detection reports are not available when the application is suspended at stop or in an abend state.

Memory-Related Errors

This section discusses the following memory-related errors that can occur in an application:

- Heap corruption
- Memory leaks
- Access errors

This section provides the background for understanding how memory corruptions and leaks happen and how software in the debugger is able to detect them.

Heap Corruption

A heap corruption occurs when an application erroneously overwrites some of the data in the heap. Heap corruption can result in data corruption, memory corruption, or both.

Note:

In this section **memory corruption** refers to any corruption of memory used by the memory management system itself as opposed to heap memory used for the application data.

When an application inadvertently uses the erroneously overwritten data in the heap, it results in **data corruption** in the application. Data corruption can lead to unpredictable program behavior.

The data corruption in the heap can lead to **memory corruption** if the corrupted data in the heap is used by memory management functions in the application to allocate, access, or deallocate memory blocks. In other words, memory corruption occurs when the corrupted datum in the heap is accessed as a pointer. Memory corruptions compromise the data integrity of the application and can result in segmentation violations if the erroneously allocated or accessed memory blocks are out of the bounds of the virtual memory of the application.

Causes for Heap Corruption

Following are some of the typical causes for heap corruption:

Double-Free

A double-free error occurs when a program attempts to free a memory block that is already freed. (Example 6 illustrates how Native Inspect detects double-frees.)

Freeing Unallocated/Uninitialized Memory

Heap corruption occurs when a program tries to free memory that is not allocated to the program. Such instances include freeing uninitialized pointers where the pointer addresses memory outside the allocated memory. (Example 7 illustrates how Native Inspect detects such errors.)

Memory Leaks

A memory leak occurs when an application fails to free allocated memory. As a result, the kernel frees the memory that is allocated by a process only when the process terminates. If the program leaks memory on a continual basis, the virtual memory requirement for the process continues to increase and this can result in serious consequences for long-running applications and memory intensive applications.

Memory leaks can also cause fragmentation of the heap. This slows down the allocation, de-allocation, and access of memory blocks and can eventually cause the application to fail with out-of-memory errors.

When to Suspect a Memory Leak

You must suspect a memory leak in an application if the system runs to its allocated limit, runs slower, or both. Memory leaks in an application increase the memory consumption in an application. When the memory consumed by the application exceeds the resource limits set by the kernel, the application fails with out-of-memory errors.

Native Inspect enables you to detect out-of-memory conditions through runtime memory checking. It also enables you to simulate out-of-memory conditions in an application to understand application behavior under such conditions.

Types of Memory Leaks

Following are the types of memory leaks:

Physical Leaks

A physical leak is a definite memory leak that occurs when an application loses all handles, or all pointers to the allocated memory. If a valid pointer to a memory block is absent, the elusive block of memory can not be accessed or freed.

The handles to a memory block are typically lost under the following conditions:

- When an application overwrites a pointer that addresses a block of memory with another address or data
- When a pointer variable goes out of scope
- When you free a structure or an array that has pointers which are not freed

When all handles to a block of memory are lost, it causes the block to be leaked. Example 9, Example 10, and Example 11 illustrate how Native Inspect detects memory leaks.

Logical Leaks

A logical leak occurs when an application fails to optimally utilize the allocated memory. In this case the allocated block of memory can still be accessed through a pointer variable in the application.

The typical causes for logical leaks are listed below:

- Leaks caused by premature allocation of memory. The application allocates the memory much ahead of the actual use of the allocated memory
- Leaks caused by delayed de-allocation
- The application delays the freeing the allocated block beyond the actual use of the allocated memory leaks caused by failure to utilize allocated memory

The application allocates memory, but fails to use the allocated memory.

Note:

Native Inspect supports the debugging of physical memory leaks only. It does not detect logical memory leaks.

Access Errors

Memory access errors can occur under the following conditions:

- When reading uninitialized local or heap data
 - When reading or writing to nonexistent, unallocated, or unmapped memory
 - When a stray pointer overflows the bounds of a heap block, or tries to access a heap block that is already freed to cause buffer overruns and under-runs
 - When reading or writing to memory locations that are already freed in the program
-

Note:

Currently, Native Inspect does not provide support for debugging memory access errors.

Using Native Inspect to Debug Memory Problems

Native Inspect supports the memory-debugging of applications involving dynamic allocations and de-allocations of virtual memory blocks, or during the calls to `zcredll` string routines like `strcpy()`, and `memcpy()`. It debugs memory-related problems at the time of allocation or de-allocation of memory blocks. It supports the detection of outstanding memory-related problems at specific user-defined probe-points (breakpoints) while heap memory is being used.

Note:

The term probe-point is used subsequently to refer to a program execution path location at which the program is suspended for the purpose of making a heap check.

Memory-related problems that appear after the specified probe points are not detected. Native Inspect does not support the debugging of access errors that are caused when reading from or writing to unallocated, uninitialized, or de-allocated memory. Native Inspect does not support the memory-debugging of the stack, static memory, or register memory.

Native Inspect provides the interactive and attach modes for debugging memory-related problems. See Supported Modes of Memory-debugging in Native Inspect for more information on the supported modes for debugging.

Memory-Debugging Features of Native Inspect

Native Inspect supports the following memory-debugging features:

- Heap Profiling features
- Leak Profiling feature
- Event Monitoring features

In addition to these features, Native Inspect provides the following generic commands for memory debugging:

Table 1: Generic Commands for Memory Checking

Command	Arguments	Command Description
<code>set heap-check</code>	<code><on/off></code>	Enables and disables memory checking. This includes the setting of commands for detecting leaks, bounds, double frees, and heap profiling.
<code>show heap-check</code>		Displays the current settings for memory checking

Heap Profiling

You can profile the heap usage in an application by using Native Inspect. The heap-profiling feature enables you to analyze the influence of algorithms and data structures on heap usage and tune the memory requirements of an application.

The point-in-time (meaning the application is suspended at some point in execution) profile displays the outstanding heap allocations at a specific instant (probe point) at runtime. It does not display the blocks that are already freed before the probe point. Native Inspect supports this point-in time heap-analysis type profile of memory use.

Note:

Heap profiling must be enabled to view heap reports.
The `set heap-check on` command enables heap profiling.

Table 2: Commands for Heap Profiling

Command	Description
<code>info heap</code>	Displays the heap report that includes the current heap allocations, the sizes of the blocks allocated, and number of allocation instances.
<code>info heap <filename></code>	Writes the heap report output to the specified file.
<code>info heap <idnumber></code>	Displays detailed information about the specified heap allocation including the allocation call stack.
<code>set heap-check min-heap-size <num></code>	Reports the heap allocations that exceed the specified number, <code><num></code> , of bytes based on the cumulative number of bytes that are allocated at each call-site inclusive of multiple calls to <code>malloc()</code> at a particular call site. See Example 1 for more information.

To obtain a point-in-time heap profile, complete the following steps:

1. Run the debugger and load the program by entering the following command at command prompt:

```
> rund <executable>/lib $system.sys00.zrtcdll/ <arguments>
(eInspect 0,105): set heap-check on
```

And for OSS:

```
: run -debug -lib /G/system/sys00/zrtcdll <arguments>
(eInspect 0,105): set heap-check on
```

Note:

The `set heap-check on` command enables the memory-debugging feature in Native Inspect. This enables the detection of leaks, heap profiles, bounds checking, checking for double free.

2. Set a breakpoint by entering the following command:

```
(eInspect 0,105): b <probepoint>
```

3. Run the program by entering the following command:

```
(eInspect 0,105):continue
```

4. When the program is stopped at a breakpoint, enter the following info heap command:

```
(eInspect 0,105): info heap
```

The following output is displayed:

```
Analyzing heap ...
Actual Heap Usage:
    Heap Start      =      0x08001000
    Heap End        =      0x0843d000
    Heap Size       =      4440064 bytes
Outstanding Allocations:
    50769 bytes allocated in 1011 blocks

No.   Total bytes   Blocks   Address           Function
0     49000          1000    0x08434cb0       main()
[...]
```

5. To view a specific allocation, specify the allocation number as an argument to the info heap command.

For example:

```
(eInspect 0,105): info heap 1
4096 bytes at 0x7bd63000 (9.86% of all bytes allocated)
in h_func () at testc:108
in main () at testc:17
in _start ()
in $START$ ()
```

You can control the stack frames that are collected for reporting at any allocation point. For more information on this feature, see “Settings to Manage Performance Degradation”

Example 1 illustrates the use of the `info heap` command with the `min-heap-size` filter setting.

Example 1: Filtered Heap Reporting for Allocations Exceeding <num> at a Particular Call-Site

Sample Program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void set_brkpt_here(int i) { }
4  main()
5  {
6      int i, *arr[1000];
7      for (i=0; i < 1000; i++)
8          arr[i] = malloc (49);
9          malloc (30);
10         set_brkpt_here(0);
11         exit(0);
12     }
13 }
```

Sample Debugging Session

```
(eInspect 1,582): b set_brkpt_here
Breakpoint 2 at 0x70000a00:0: file \PELICAN.$DATA4.USER.EX1C, line 3.
(eInspect 1,582): set heap-check min-heap-size 31
(eInspect 1,582): c
Continuing.

Breakpoint 2, set_brkpt_here (i=138583280) at \PELICAN.$DATA4.USER.EX1C:3
*   3          void set_brkpt_here(int i) { }
(eInspect 1,582): info heap
Analyzing heap ...

Actual Heap Usage:
    Heap Start      =      0x08001000
    Heap End        =      0x0843d000
    Heap Size       =      4440064 bytes

Outstanding Allocations:
    50769 bytes allocated in 1011 blocks

No.   Total bytes   Blocks   Address           Function
0     49000          1000    0x08434cb0        main()
1     1024           1       0x08404560        <system/unknown>()
```

Note:

The "<system/unknown>" function displayed indicates memory allocated from an unknown function (function with no name) or system routine.

Leak Profiling

The leak profile feature in Native Inspect conservatively identifies the blocks of memory that are leaked in an application, and displays the stack trace that shows when the block was allocated. All the leaks detected by Native Inspect are definite physical leaks.

Native Inspect uses a garbage collection algorithm to identify the blocks that are leaked. It identifies the root-set of memory that indicate possible pointers to the heap. The initial root-set includes the shared library data, the program stack, the registers. The initial root-set includes all data except the heap blocks.

The debugger considers suitably aligned words in the root-set as possible pointers to the heap. The debugger performs a reachability analysis based on the root-set, and determines the memory blocks that are reachable through possible pointers from the root-set. The heap blocks that are not reachable through possible pointers from the root-set are reported as leaks.

Native Inspect is conservative in detecting the memory leaks. The memory leaks can be masked if a datum in the root-set inadvertently holds a possible pointer to a heap block. An example of “inadvertently” would be where a pointer value is held in a floating point variable instead of one that is declared “pointer to”.

Table 3: Commands for Leak Profiling

Command	Description
<code>set heap-check leaks <on/off></code>	Controls Native Inspect memory leak checking.
<code>info leaks</code>	Displays a leak report. It also lists information such as the leaks, size of blocks, and number of instances.
<code>info leaks <filename></code>	Writes the complete leak report output to the specified file. If a filename is given, the output produced in the file is a detailed output. Meaning the trace of each leak is also provided.
<code>info leak <leaknumber></code>	Displays detailed information on the specified leak including the allocation call stack.
<code>set heap-check min-leak-size <num></code>	Specifies the minimum leak size for stack trace collection. The debugger continues to report leaks that are smaller than <num> bytes, but it does not provide the stack trace for the same. By default, <code>num</code> is set to 0. This command also enables you to reduce performance degradation. See “Settings to Manage Performance Degradation.”

To view the leak profile, complete the following steps:

1. Run the debugger and load the program by entering the following command:

```
> rund <executable>/debug, lib $system.sys00.zrtcdll/ <arguments>
```

2. Enable leak checking by entering the following command:

```
(eInspect 0,105): set heap-check leaks on
```

Note:

Alternatively, you can use the `set heap-check on` command to automatically enable the detection of leaks by toggling the `set heap-check leaks on` command. This command enables the detection of leaks, heap profiles, bounds checking, and checking for double frees.

3. Set breakpoints in the code at probe-points where you want to examine cumulative leaks by entering the following command:

```
(eInspect 0,105): b <probe-points>
```

4. Continue the program in the debugger by entering the following command:

```
(eInspect 0,105): c
```

5. When the breakpoint triggers, enter the following info leaks command to display the list of memory leaks:

```
(eInspect 0,105): info leaks
```

The following output is displayed:

```
Scanning for memory leaks...done
2439 bytes leaked in 25 blocks
No. Total bytes Blocks Address Function
0 1234 1 0x08434ac0 func1()
1 333 1 0x08434cb0 main()
2 245 8 0x08434da0 strdup()
[...]
```

The debugger assigns a numeric identifier for each leak. To view a stack trace for a specific leak, specify the leak number from the list of leaks, as follows:

```
(eInspect 0,105): info leak 2
245 bytes leaked in 8 blocks (10.05% of all bytes leaked)
These range in size from 26 to 36 bytes and are allocated in strdup ()
in link_the_list () at testc:55
in main () at testc:13
in _MAIN ()
```

Event Monitoring

The event monitoring commands in Native Inspect enable you to monitor specific heap events and heap-corruption problems in an application.

Monitoring Heap Events

Native Inspect enables you to monitor specific events such as the size of memory allocations.

Table 4: Monitoring Heap Events

Command	Description
<code>set heap-check block-size <num-bytes></code>	Suspends program execution when a program tries to allocate a block larger than num-bytes in size.
<code>set heap-check heap-size <num-bytes></code>	Suspends program execution when the program tries to increase the program-heap by at least num-bytes.
<code>set heap-check free <on off></code>	Toggles the detection of double-frees and frees with improper arguments.

Monitoring Allocations Greater Than a Specified size

The `set heap-check block-size` command instructs Native Inspect to stop the program and transfer the execution control to the user when the program allocates a heap block whose size is greater than or equal to `<num-bytes>`. Following is the syntax for the `set heap-check block-size` command:

```
set heap-check block-size <num-bytes>
```

Example 2: Monitoring allocations greater than a specified size

Sample Program

```
1      #include <stdio.h>
2
3      int main()
4      {
5          char * cp;
6          printf("Start of the program\n");
7          cp = (char *)malloc(1024 *1024*10);
8          free (cp);
9          exit(0);
10     }
```

Sample Debugging Session

```
(eInspect 1,575): set heap-check on
(eInspect 1,575): set heap-check block-size 900000
(eInspect 1,575): b 9
Breakpoint 2 at 0x70000b00:1: file \PELICAN.$DATA4.USER.EX2C, line 9.
(eInspect 1,575): c
Continuing.
warning: Attempt to allocate a large object at 0x0842ad28
__rtc_event (ecode=RTC_HUGE_BLOCK, pointer=0x842ad28, pclist=0x0, size=0)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901: Guardian or User Defined
Error 13
(eInspect 1,575): bt
#0  __rtc_event (ecode=RTC_HUGE_BLOCK, pointer=0x842ad28, pclist=0x0,
size=0)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901
#1  0x78008120:0 in rtc_record_malloc (pointer=0x842ad28 "",
size=10485760,
  heap_block=1, padded=1)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:3158
#2  0x7800ae60:0 in malloc (size=10485760)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:3912
#3  0x70000aa0:0 in main () at \PELICAN.$DATA4.USER.EX2C:7
#4  0x70000cc0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
(eInspect 1,575): c
Continuing.

Breakpoint 2, main () at \PELICAN.$DATA4.USER.EX2C:9
*      9          exit(0);
```

Monitoring De-allocations to Detect Double-Frees

The `set heap-check free <on/off>` command enables you to detect double-frees and frees with improper arguments. When this command is enabled, the `free()` calls are monitored to verify whether the parameters address valid heap blocks. If an erroneous `free()` is detected, the debugger stops execution and reports the error. You can analyze the stack trace to analyze where and how the error occurred.

Example 6 illustrates the use of the `set heap-check free` command.

Monitoring Heap Corruption

Native Inspect enables you to detect the presence of heap-corruption in your application.

Table 5: Commands for Monitoring Heap Corruption

Command	Description
<code>set heap-check string <on/off></code>	Toggles validation of calls to <code>strcpy()</code> , <code>strncpy()</code> , <code>memcpy()</code> , <code>memccpy()</code> , <code>memset()</code> , <code>memmove()</code> , <code>bzero()</code> , <code>strdup()</code> , and <code>bcopy()</code>
<code>set heap-check bounds <on/off></code>	Toggles the bounds-checking feature for detection of heap-corruption in Native Inspect
<code>info corruption</code>	Displays a list of all the dangling pointers and dangling blocks that are potential sources of memory corruption

Finding the precise line where heap corruption occurs requires some more effort. There are several strategies to find this point. If an address in heap space is being overwritten, the user can restart the application and then place a MAB, memory access breakpoint, using the suspected address after the point of allocation. Also, the application could be restarted then run to the allocation point for a corrupted block of memory. Then a process of binary searches could be performed to find the precise point of corruption. The searching requires finding a good mid-point of execution between the lines where you know the block is okay and where the `info corruption` command reported a failure. If the corrupt block is a character string, the following section may help with information on another method of finding the problem.

Monitoring String Corruption

The `set heap-check string <on/off>` command toggles the string corruption detection feature. It enables you to detect string corruption if functions of the `strcpy()` family write out-of-bounds of the allocated memory. Example 3 illustrates the use of the `set heap-check string` command.

This command currently detects string corruption when writing out-of-bounds for `strcpy()`, `strncpy()`, `memcpy()`, `memccpy()`, `memset()`, `memmove()`, `bzero()`, `strdup()`, and `bcopy()` functions.

Example 3: Monitoring heap-corruption caused by erroneous handling of string functions

Sample Program

```
1   #include <stdio.h>
2
3   int main()
4   {
5       char *ptr, *ptr1;
6
7       ptr = (char*)malloc(10);
8       ptr1 = (char *)malloc(20);
9
10      strcpy(ptr, "Hello");
11      strcpy(ptr1, "Welcome to Native Inspect");
12
13      memcpy(ptr+5, ptr1, 10);
14  }
```

Sample Debugging Session

```
(eInspect 1,1193): set heap-check on
(eInspect 1,1193): set heap-check string on
(eInspect 1,1193): c
Continuing.
warning: strcpy corrupted (address = 0x08429d18 size = 20)
#1 main() at \PELICAN.$DATA4.USER.EX3C:8
#2 _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of eInspect.

__rtc_event (ecode=RTC_BAD_STRCPY, pointer=0x8429d18, pclist=0x8404a4c,
size=20)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:1901: Guardian or User Defined
Error 13
(eInspect 1,1193): bt
#0 __rtc_event (ecode=RTC_BAD_STRCPY, pointer=0x8429d18,
pcclist=0x8404a4c,
size=20)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:1901
#1 0x7800cbc0:0 in search_addr (pointer=0x8429d18 "", len=26,
this_c_i=0x840dac4, ecode=RTC_BAD_STRCPY)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:4391
#2 0x7800d150:0 in libc_mem_common (addr=0x8429d18, len=26,
ecode=RTC_BAD_STRCPY)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:4556
#3 0x7800e510:0 in strcpy (d=<optimized out>, s=<optimized out>)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:4810
#4 0x70000b60:0 in main () at \PELICAN.$DATA4.USER.EX3C:11
#5 0x70000dc0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
(eInspect 1,1193): c
Continuing.
warning: memcpy corrupted (address = 0x08429cfd size = 10)
#1 main() at \PELICAN.$DATA4.USER.EX3C:7
#2 _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of eInspect.

__rtc_event (ecode=RTC_BAD_MEMCPY, pointer=0x8429cfd, pclist=0x8404a38,
size=10)
  at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\..\Src/gnu/gdb\infrtc.c:1901: Guardian or User Defined
Error 13.
```

Detecting Out-of-Bounds Writes with the Bounds-Checking Feature

The `set heap-check bounds <on/off>` command toggles the bounds-checking feature in Native Inspect. When bounds-checking is enabled, Native Inspect allocates extra space (guard bytes) at the beginning and end of a block during allocation and fills this space with a specific pattern. When the blocks are freed, the debugger verifies if the patterns are intact. If the patterns are corrupted, the debugger detects underflow or overflow errors and reports the corruption. Example 4 illustrates the bounds-checking feature.

The bounds-checking feature detects overflow and underflow errors only when the write operation occurs within the guard bytes.

Example 4: Bounds-checking to detect out-of-bounds writes

Sample Program

```
1      #include <stdio.h>
2
3      int main()
4      {
5          char *cp = (char*)malloc(100);
6          cp[-1] = 100;
7          strcpy(cp, "Hello");
8          cp[100] = 100;
9          free(cp);
10         exit(0);
11     }
```

Sample Debugging Session

```
(eInspect 1,558): set heap-check bounds on
(eInspect 1,558): b 10
Breakpoint 2 at 0x70000c10:1: file \PELICAN.$DATA4.USER.EX4C, line 10.
(eInspect 1,558): c
Continuing.
warning: Memory block (size = 100 address = 0x08429cf8) appears to be
corrupted at the beginning.
Allocation context might not be correct.

#1  main() at \PELICAN.$DATA4.USER.EX4C:5
#2  _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
warning: Use command backtrace (bt) to see the current context.

Ignore top 4 frames belonging to leak detection library of eInspect.

__rtc_event (ecode=RTC_BAD_HEADER, pointer=0x8429cf8, pclist=0x8404a38,
size=100)
    at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src\gnu\gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src\gnu\gdb\infrtc.c:1901: Guardian or User Defined
Error 13
(eInspect 1,558): c
Continuing.

Breakpoint 2, main () at \PELICAN.$DATA4.USER.EX4C:10
*   10          exit(0);
```

The size of guard bytes for every block of the allocated memory is currently set to 8 leading (header) bytes and 1 tail (footer) byte. These guard bytes are used to detect boundary (buffer over-run and buffer under-run) memory corruptions.

Detecting Heap Corruption

The `info corruption <filename>` command enables you to view the corruption profile of all the allocations that are corrupted at a specified probe-point in the program. Ensure that the bounds checking is enabled before using the `info corruption` command. The corruption information is written to a specified file if the `<file name>` is provided. Otherwise, it is written to `stdout`.

Example 5 Detecting heap corruption using the `info corruption` command

Sample Program

```
1      #include <stdio.h>
2      #include <stdlib.h>
3
4      char *t;
5      char *t1;
6      char *t2;
7      char *t3;
8
9      char * sm_malloc(sz)
10     int sz;
11     {
12         return (char *)malloc(sz); /* line number 12 */
13     }
14
15     main()
16     {
17         t = (char *)sm_malloc(10);
18         strcpy(t, "123456789123");
19         t1 = (char *)sm_malloc(10);
20         strcpy(t1, "12345678912");
21         t2 = (char *)sm_malloc(10);
22         strcpy(t2, "1234567891");
23         t3 = (char *)sm_malloc(10);
24         strcpy(t3, "123456789");
25         printf("Hello\n");
26         free (t);
27         free (t1);
28         free (t2);
29         free (t3);
30         free (t);
31         free (t1);
32         exit(1);
33     }
```

Sample Debugging Session

```
(eInspect 1,666): b 25
Breakpoint 2 at 0x70000e00:1: file \PELICAN.$DATA4.USER.EX5C, line 25.
(eInspect 1,666): set heap-check on
(eInspect 1,666): c
Continuing.

Breakpoint 2, main () at \PELICAN.$DATA4.USER.EX5C:25
* 25          printf("Hello\n");
(eInspect 1,666): info corruption
Analyzing heap ...

Following blocks appear to be corrupted
No.   Total bytes   Blocks   Corruption   Address   Function
0     10            1       End of block 0x08429cf8 sm_malloc()
1     10            1       End of block 0x08429d18 sm_malloc()
2     10            1       End of block 0x08429d38 sm_malloc()
(eInspect 1,666): info corruption 2
10 bytes at 0x08429d38 (33.33% of all bytes allocated)
#0  sm_malloc() at \PELICAN.$DATA4.USER.EX5C:12
#1  main() at \PELICAN.$DATA4.USER.EX5C:21
#2  _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAINC:46
```

Settings to Manage Performance Degradation

Memory-debugging slows down the performance of an application by roughly 2-3 times because of stack unwinding. Reducing the number of stack frames the debugger collects for each allocation reduces the performance degradation.

Table 6: Options for Performance Improvement

Command	Setting	Description
<code>set heap-check frame-count <num></code>	By default, <code>num</code> is set to 4	Controls the depth of the call stack.
<code>set heap-check min-leak-size <num></code>	By default, <code>num</code> is set to 0.	Specifies the minimum leak size for stack trace collection. The debugger continues to report leaks that are smaller than <code><num></code> bytes, but it does not provide the stack trace for the same.

Supported Modes of Memory-debugging in Native Inspect

Native Inspect supports the following modes of memory-debugging:

- Interactive mode (debugging programs launched under debugger control)
- Attach mode

Debugging Programs Launched Under Debugger Control

The interactive mode of memory-debugging is typically useful during the development and defect fixing phase, where you need the flexibility to control the flow of program execution while debugging memory related problems.

To debug your program in the interactive mode, complete the following steps:

1. Compile the source files with the `symbols` or `-g` option. No special compilation or link options are required.
2. To activate the memory debugging, perform either of the following:

Invoke Native Inspect with the `lib` option as follows:

```
> rund <executable> /lib $system.sys00.zrtcdll/
```

This enables leak checking. To enable other memory debugging features, you must use the appropriate set of commands.

Or, on OSS, enter the following command:

```
: run -debug -lib=/G/system/sys00/zrtcdll <executable>  
(eInspect 0,105):set heap-check on
```

This enables leaks checking, bounds checking, and check for double-frees.

3. Place breakpoints at probe points by entering the following command:

```
(eInspect 0,105):b <probe_point>
```

4. To generate a leak profile at the breakpoint, enter the following command:

```
(eInspect 0,105):info leaks
```

5. To generate a point-in-time heap profile at the breakpoint, enter the following command:

```
(eInspect 0,105): info heap
```

Either command's output may be logged into a file. For example, here is how to save the output from the `leaks` command:

```
(eInspect 0,105):info leaks <filename>
```

Debugging a Running Process

Native Inspect can attach to a running process and debug memory problems. However, to use the debugger in this mode, the application must be launched using the user library option. To debug memory while attaching Native Inspect to a running process, complete the following steps:

1. Run the executable with the `lib` option.
2. Identify the required process (for example, by using the `status` or `ps` command) and attach the debugger to the process as follows (where `N` is the same `cpu` used to run the executable).

```
> einspect /cpu N/  
(einspect 0,105) attach <process-id>
```

3. Insert breakpoints at suitable probe-points. When the breakpoints trigger, use the `info heap` and `info leaks` commands to display the heap and leak profile.

Note:

An application running under the PATHWAY server is not compatible with Native Inspect memory leak detection.

Summary of Memory Debugging Commands

Table 7: Commonly Used Commands for Memory Debugging

Description	Command
Enables heap profiling	<code>set heap-check <on/off></code>
Enables you to detect leaks	<code>set heap-check leaks <on/off></code>
Enables you to detect double-frees and frees with improper arguments	<code>set heap-check free <on/off></code>
Enables you to check for out-of-bounds corruption when the block is freed	<code>set heap-check bounds <on/off></code>
Enables validation of calls to <code>strcpy()</code> , <code>strncpy()</code> , <code>memcpy()</code> , <code>memccpy()</code> , <code>memset()</code> , <code>memmove()</code> , <code>bzero()</code> , <code>strdup()</code> , and, <code>bcopy()</code>	<code>set heap-check string <on/off></code>
Enables you to set the number of frames to be printed for leak and heap profiles	<code>set heap-check frame-count <num></code>
Enables you to set the minimum block size to report in heap profiles	<code>set heap-check min-heap-size <num></code>
Enables you to set the minimum block size to use for leak detection	<code>set heap-check min-leak-size <num></code>

Conclusion

Memory-related errors are some of the most difficult programming errors to detect and debug. Debugging memory-related errors is difficult without the help of an effective memory analysis tool. Native Inspect enables you to debug memory leaks and heap-related errors in an application. In addition to plugging memory leaks in your application, it is also important to track the memory utilization in your application. Native Inspect provides capabilities such as heap profiling to analyze the memory-usage of your application. The heap profile displays information about the allocated memory, the calling function, and it also displays the allocating call stack.

Additional Examples

Example 6 to Example 11 illustrate how Native Inspect detects memory leaks and heap-errors caused by different types of programming errors.

Example 6 Detecting a double-free error

Sample Program

```
1      #include <stdio.h>
2
3      int main()
4      {
5          char* han;
6          printf("Starting program\n");
7          han = (char*)malloc(sizeof(char));
8          free(han);
9          printf("Now freeing a pointer twice...\n");
10         free(han);
11     }}
```

Sample Debugging Session

```
(eInspect 1,1180): set heap-check free on
(eInspect 1,1180): n
*   7          han = (char*)malloc(sizeof(char));
(eInspect 1,1180): n
*   8          free(han);
(eInspect 1,1180): n
*   9          printf("Now freeing a pointer twice...\n");
(eInspect 1,1180): n
*  10          free(han);
(eInspect 1,1180): n
warning: Attempt to free unallocated or already freed object at
0x0842ad10
__rtc_event (ecode=RTC_BAD_FREE, pointer=0x842ad10, plist=0x0, size=0)
at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src\gnu\gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src\gnu\gdb\infrtc.c:1901: Guardian or User Defined
Error 13
```

Example 7 Detecting de-allocation of memory that has not been initialized

Sample Program

```
1      #include <stdio.h>
2
3      int main() {
4          char* han = "HI";
5          printf("Starting program\n");
6
7          free(han);
8     }
```

Sample Debugging Session

```
(eInspect 1,700): set heap-check on
(eInspect 1,700): n
*   5           printf("Starting program\n");
(eInspect 1,700): n
*   7           free(han);
(eInspect 1,700): n
warning: Attempt to free unallocated or already freed object at
0x080001a0
__rtc_event (ecode=RTC_BAD_FREE, pointer=0x80001a0, pclist=0x0, size=0)
   at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901: Guardian or User Defined
Error 13
```

Example 8 Detecting de-allocation of unallocated blocks

Sample Program

```
1   #include <stdio.h>
2
3   int main() {
4       int *han;
5       printf("Starting program\n");
6       han = (int*)malloc(sizeof(int));
7       han++;
8       free(han);
9   }
```

Sample Debugging Session

```
(eInspect 1,677): set heap-check on
(eInspect 1,677): n
*   6           han = (int*)malloc(sizeof(int));
(eInspect 1,677): n
*   7           han++;
(eInspect 1,677): n
*   8           free(han);
(eInspect 1,677): n
warning: Attempt to free unallocated or already freed object at
0x0842ad2c
__rtc_event (ecode=RTC_BAD_FREE, pointer=0x842ad2c, pclist=0x0, size=0)
   at c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901
c:\t1237\h0616\wdb\build\ia64-hp-nsk-
native\gdb\..\..\Src/gnu/gdb\infrtc.c:1901: Guardian or User Defined
Error 13
```

Example 9 Detecting memory leaks caused when an application overwrites a pointer that currently addresses a block of memory with another address or data

Sample Program

```
1      #include <stdio.h>
2
3      int main() {
4          int* han1, * han2;
5          printf("Starting program\n");
6          han1 = (int*)malloc(sizeof(int));
7          han2 = (int*)malloc(sizeof(int));
8          han1 = han2;
9          free(han1);
10     }
```

Sample Debugging Session

```
(eInspect 1,699): set heap-check on
(eInspect 1,699): n
* 6          han1 = (int*)malloc(sizeof(int));
(eInspect 1,699): n
* 7          han2 = (int*)malloc(sizeof(int));
(eInspect 1,699): n
* 8          han1 = han2;
(eInspect 1,699): n
* 9          free(han1);
(eInspect 1,699): n
* 10         }
(eInspect 1,699): info leak
Scanning for memory leaks...

4 bytes leaked in 1 blocks

No.   Total bytes   Blocks   Address   Function
0      4             1       0x0842ad28  main()
(eInspect 1,699): info leak 0
4 bytes leaked at 0x0842ad28 (100.00% of all bytes leaked)
#0  main() at \PELICAN.$DATA4.USER.EX9C:6
#1  _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
```

Example 10 Detecting memory leaks that are caused when a pointer variable in an application addresses memory that is out of the scope of the application

Sample Program

```
1      #include <stdio.h>
2
3      void func1(int* ptr1)
4      {
5          ptr1 = (int*)malloc(5*sizeof(int));
6      }
7
8      void func2(int** ptr)
9      {
10         func1(*ptr);
11     }
12     int main()
13     {
14     {
15         int* han1;
16         printf("Starting program\n");
17         func2(&han1);
18         printf("End of the program\n");
19     }
20     }
```

Sample Debugging Session

```
(eInspect 1,707): set heap-check on
(eInspect 1,707): b 18
Breakpoint 2 at 0x70000ce0:0: file \PELICAN.$DATA4.USER.EX10C, line 18.
(eInspect 1,707): c
Continuing.

Breakpoint 2, main () at \PELICAN.$DATA4.USER.EX10C:18
* 18         printf("End of the program\n");
(eInspect 1,707): n

* 19         }
(eInspect 1,707): info leak
Scanning for memory leaks...

20 bytes leaked in 1 blocks

No.    Total bytes    Blocks    Address    Function
0      20              1        0x0842ad28  func1()
(eInspect 1,707): info leak 0
20 bytes leaked at 0x0842ad28 (100.00% of all bytes leaked)
#0  func1() at \PELICAN.$DATA4.USER.EX10C:5
#1  func2() at \PELICAN.$DATA4.USER.EX10C:11
#2  main() at \PELICAN.$DATA4.USER.EX10C:18
#3  _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
```

Example 11 Detecting memory leaks when you free a structure or an array that has pointers which are not freed

Sample Program

```
1    #include <stdio.h>
2
3    struct stud
4    {
5        char* name;
6        int id;
7    };
8
9    int main() {
10
11        struct stud *s1;
12        s1 = (struct stud*)malloc(sizeof(struct stud));
13        s1->name = (char*)malloc(50);
14        strcpy(s1,"Annie");
15        s1->id=10;
16        free(s1);
17    }
```

Sample Debugging Session

```
(eInspect 1,712): set heap-check on
(eInspect 1,712): b 17
Breakpoint 2 at 0x70000b60:1: file \PELICAN.$DATA4.USER.EX11C, line 17.
(eInspect 1,712): c
Continuing.

Breakpoint 2, main () at \PELICAN.$DATA4.USER.EX11C:17
* 17          }
(eInspect 1,712): info leak
Scanning for memory leaks...

50 bytes leaked in 1 blocks

No.   Total bytes   Blocks   Address   Function
0     50           1       0x08429d18  main()
(eInspect 1,712): info leak 0
50 bytes leaked at 0x08429d18 (100.00% of all bytes leaked)
#0  main() at \PELICAN.$DATA4.USER.EX11C:13
#1  _MAIN() at \SPEEDY.$RLSE.T8432H02.CPLMAIN:46
```

FAQ

1. Does Native Inspect report all the leaks in a program?

Native Inspect uses a conservative leak detection algorithm. As a result, all leaks may not be reported, but all reported leaks are definite leaks. Native Inspect reports leaks only in the code path exercised in the current run.

2. I wrote a small sample program that allocates a block using `malloc()` and leaks the block immediately, by assigning `NULL` to the pointer, but Native Inspect does not report this block as a leak. Why?

This is attributed to the leak detection algorithm followed by Native Inspect. If the datum in the program address space masks a leak, the leak is not reported. In this case the address returned from `malloc()` is stored in the architecture registers and consequently masks the leak. Typically, if you call any function after the leak, such as a `printf()`, Native Inspect can catch the leak.

3. Does Native Inspect support detection of leaks in third-party code?

Yes. Native Inspect supports detection of leaks in third-party code also.

4. Can Native Inspect debug applications with user-defined memory management routines?

Native Inspect can debug applications with memory management routines that are either user defined or are wrappers to the default memory management routines. However, the granularity of tracing memory failures may be reduced to only the sections of memory allocated through `malloc()`. Other smaller blocks allocated within this region will not be memory checked.

5. Does Native Inspect report the exact instant when the block becomes a leak?

No. Native Inspect does not provide information on when the leak occurred. It reports only the allocation stack trace of the leaked block and does not report the stack trace where the block leaked.

6. Does Native Inspect support debugging of C++ applications with calls to `new()` and `delete()`?

Yes. Native Inspect supports debugging of C++ applications with `new()` and `delete()` calls, but only if they internally call `malloc()` and `free()`.

7. Does Native Inspect support memory-debugging of long running applications?

Yes. Native Inspect supports debugging of long running applications as long as the memory leak detection library has been specified as an option when the application starts.

8. Does the debugger find leaks in the executable from the startup of the application when debugging the application in attach mode?

Yes, the debugger finds leaks in the executable from the startup of the executable by default, when debugging in attach mode.

9. When attempting to view the leak report, the following error occurs:

```
(eInspect 0,105): info leaks  
Scanning for memory leaks...  
Error downloading data !  
(eInspect 0,105):
```

What is the cause for this error and what is the work-around?

This error message is displayed when you attempt to view the heap profile or the leak profile of a debugged process, which is exiting or has exited program execution. As a work-around, you can place a breakpoint before the program exits and then enter the `info leaks` command or the `info heap` command.

10. What is the work-around if the following error message is displayed while debugging memory?

```
(eInspect 0,105): info corruption  
Current thread is inside the allocator. Try again later.
```

This error message signifies that the program execution is in a frame that belongs to a Native Inspect internal leak detection library. When this error is encountered, it is not safe to enter commands that involve calls to the leak detection library procedures. The user must set the frame to the last leak detection library frame and enter the `finish` command before resuming to debug memory.

Technology for better business outcomes

© Copyright 2008 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Linux is a U.S. registered trademark of Linus Torvalds. Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation. UNIX is a registered trademark of The Open Group.

4AA2-3323ENW, November 2008

