CMU 17-654 & 17-754 Analysis of software Artifact Jonathan Aldrich, Spring 2008

Tool Evaluation Report on Purify

Average Time Spent: *aprx. 20 hours*

Team AoB: jinheec, sjang, taekgook, gdo Due: Tue Mar. 25, 2008

1 Introduction

Typically run-time memory errors and leaks are very difficult to locate. The symptoms of incorrect memory usage are unpredictable and usually appear far from the cause of the error. IBM Rational Purify is a runtime analysis tool designed to help developers write reliable code. It is one of three related tools packaged in IBM Rational® PurifyPlus. The package includes Rational Purify, Rational Quantify and Rational PureCoverage [3]:

- Rational Purify® is an automatic error detection tool for finding runtime errors and memory leaks in every component of program.
- Rational Quantify® is a performance analysis tool for resolving performance bottlenecks so your program can run faster.
- Rational PureCoverage® is a code coverage tool for making sure your code is thoroughly tested before you release it.

Rational Purify tests a program written in C/C++, Java, C# or VB .Net and it supports Linux , Unix, and Windows platforms(except for Windows Vista). In this report, we focus on analyzing Rational Purify for Windows in terms of usability and test soundness of it.

2 How Purify Works

Basically, in order to detect various kinds of run-time memory errors, Purify monitors every byte of memory for all memory operations by adding monitoring bits; monitor memory that is not allocated, allocated but uninitialized, and freed after use but still initialized. More precisely, Purify automatically inserts verification code to the object code by parsing. Also, it maintains a table to track the status of each byte of memory. In the table, two additional bits are used to represent status of each byte of memory. The first bit keeps track whether the corresponding byte has been allocated and the second bit records whether the byte has been initialized. With the combination of two bits, purify describes four states of memory: red, yellow, green, and blue.

Figure 1 show the four states which each byte of memory can have.

Red: Purify labels heap memory and stack memory red initially. This memory is unallocated and uninitialized. Either it has never been allocated, or it has been allocated and subsequently freed. In addition, Purify inserts guard zones around each allocated block and each statically allocated data item, in order to detect array bounds errors. Purify colors these guard zones red and refers to them as red zones. It is illegal to read, write, or free red memory because it is not owned by the program.

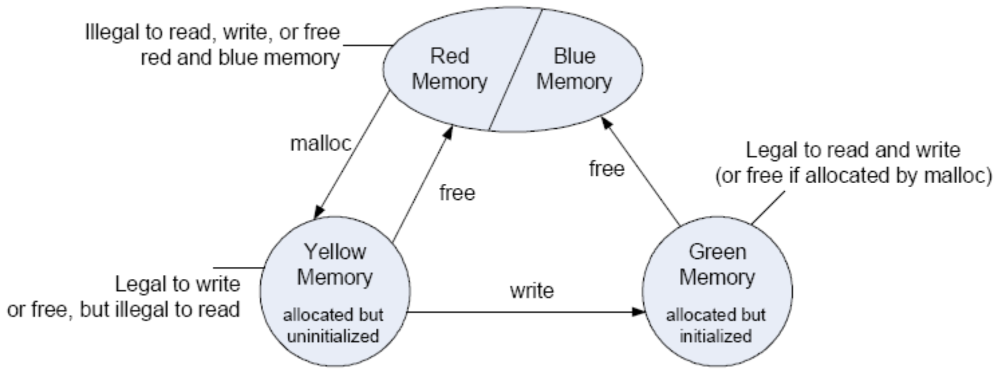

Illegal to read, write, or free red and blue memory
Red Memory
Blue Memory
malloc
free
free
Legal to read and write (or free if allocated by malloc)
Yellow Memory
allocated but uninitialized
Legal to write or free, but illegal to read
write
Green Memory
allocated but initialized

Figure 1 The status of memory in Purify

Yellow: Memory returned by `malloc` or new is yellow. This memory has been allocated, so the program owns it, but it is uninitialized. You can write yellow memory, or free it if it is allocated by `malloc`, but it is illegal to read it because it is uninitialized. Purify sets stack frames to yellow on function entry.

Green: When you write to yellow memory, Purify labels it green. This means that the memory is allocated and initialized. It is legal to read or write green memory, or free it if it was allocated by `malloc` or new. Purify initializes the *data* and *bss* sections of memory to green.

Blue: when you free memory after it is initialized and used, Purify labels it blue. This means that the memory is initialized, but is no longer valid for access. It is illegal to read, write, or free blue memory

3 Evaluation

3.1 Overall Evaluation

3.1.1 Qualitative Aspects

As mentioned in previous sections, Purify can be used to find memory-related defects. However, as much as its functionality, it is also important that Purify must fulfill to users' needs in terms of easy-to-use. We can research usability of Purify whether it meets the users' needs to use it easily. However, usability can hardly be quantified because every user might have different perspective on this matter. Thus, to normalize the qualitative issue, we would like to use a survey by the Likert scale. Because of limited time and human resource, the participants can be only four members who conduct this project, and we limit the category of usability of Purify; installation easiness, and comprehensiveness of results, error summary, and execution trace.

3.1.2 Quantitative aspects

- Performance

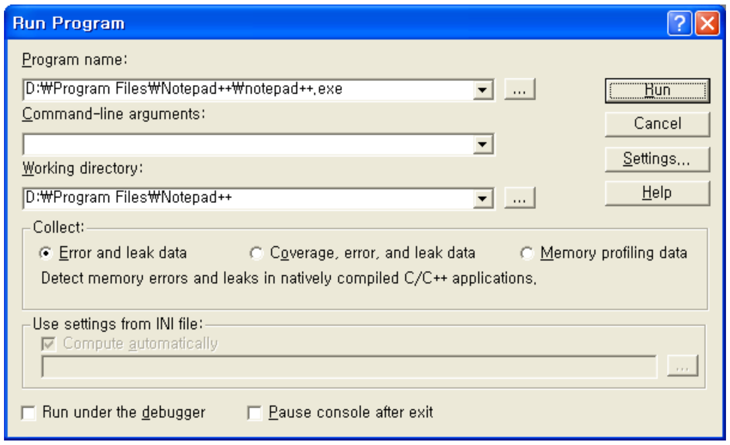
In performance, we quantified CPU and memory usage which can potentially affect on detection speed. We measured CPU and memory usage of Purify with Notepad++ and our own source code. Also, we contrasted the minutes of error detection by Purify and inspection to measure how faster Purify is than human inspection. The result of contrast was easily expected, of course; the more lines of target codes are, the faster Purify is presumably.

- Soundness

However, by contrasting two factors, we can identify the possibility of False-negative and False-positive with small chunks of codes which contains intentional defects to determine soundness of Purify. Here, the soundness of Purify is the most critical factors that we evaluate the program analysis tool. Though a particular tool provides extreme easiness of use and high performance, for example, if it detects amount of False-negative or False-positive, we might not have a trustworthy to the analysis tool. That is, it is meaningless. To measure the soundness of Purify, we conducted an experiment how well Purify detects a set of defects seeded program.

3.2 The Notepad++

To evaluate Purify, we used the Notepad++. At first time, we downloaded the latest version of Notepad++ from the Internet [2], and launched the Notepad++ using Purify. Figure 2 shows the execution dialog of Purify. As you can see, we can set which program will run and other options. If you click the "Run" button, Purify will execute the selected program to find defects.



Program name: D:₩Program Files₩Notepad++₩notepad++.exe

Command-line arguments:

Working directory: D:₩Program Files₩Notepad++

Collect: [✓ Error and leak data] Coverage, error, and leak data Memory profiling data

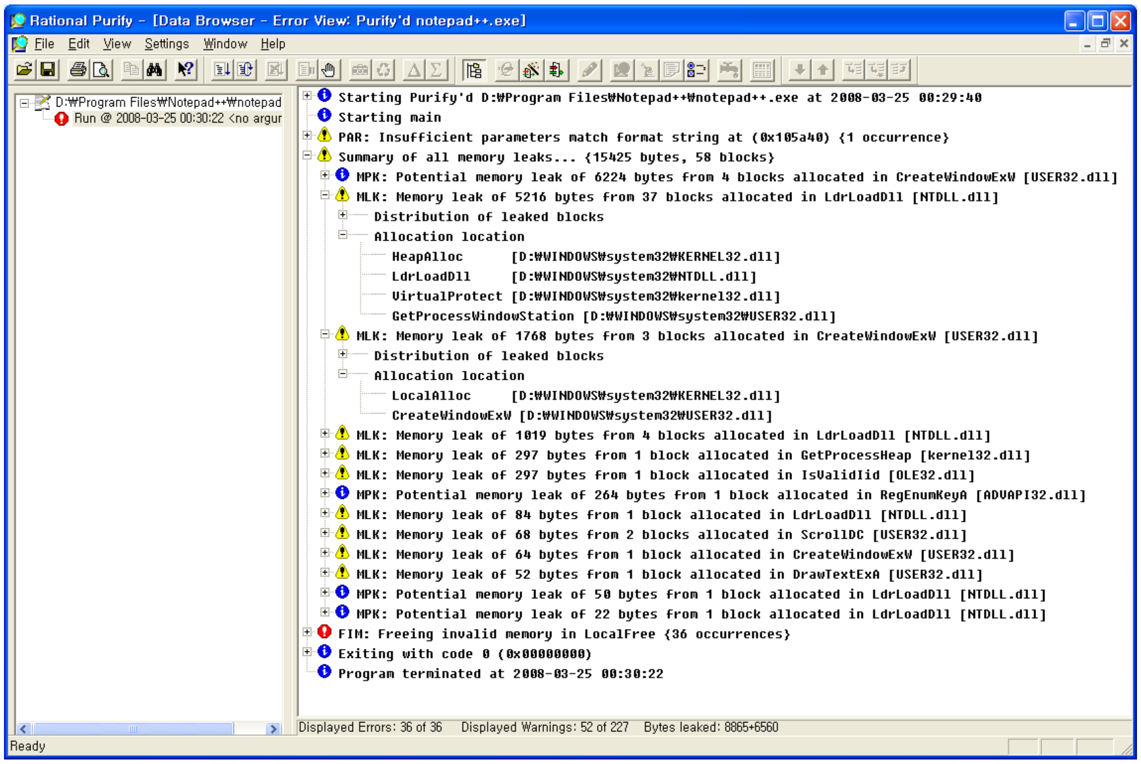
Detect memory errors and leaks in natively compiled C/C++ applications.

Use settings from INI file: [✓ Compute automatically]

☐ Run under the debugger ☐ Pause console after exit

Figure 2 Run Program dialog of Purify

Figure 3 shows the main window of Purify. The left tree view shows a list of what we tested, and the right tree view shows a list of problems which are founded by Purify. According to Purify, the Notepad++ contains some defects. Yellow exclamation marks indicate warnings, and red exclamation marks indicate errors according to the Purify manual [3]. When we click each item in the right tree view, Purify shows the related information. At this point, we faced a serious problem. It was hard to find which source code makes defects because normally an execution file does not contain debug information. As Figure 3 shows, Purify can notify only types of errors and filenames, not exact information.


Rational Purify - [Data Browser - Error View: Purify'd notepad++.exe]
File Edit View Settings Window Help
D:₩Program Files₩Notepad++₩notepad
Run @ 2008-03-25 00:30:22 <no argur
Ready

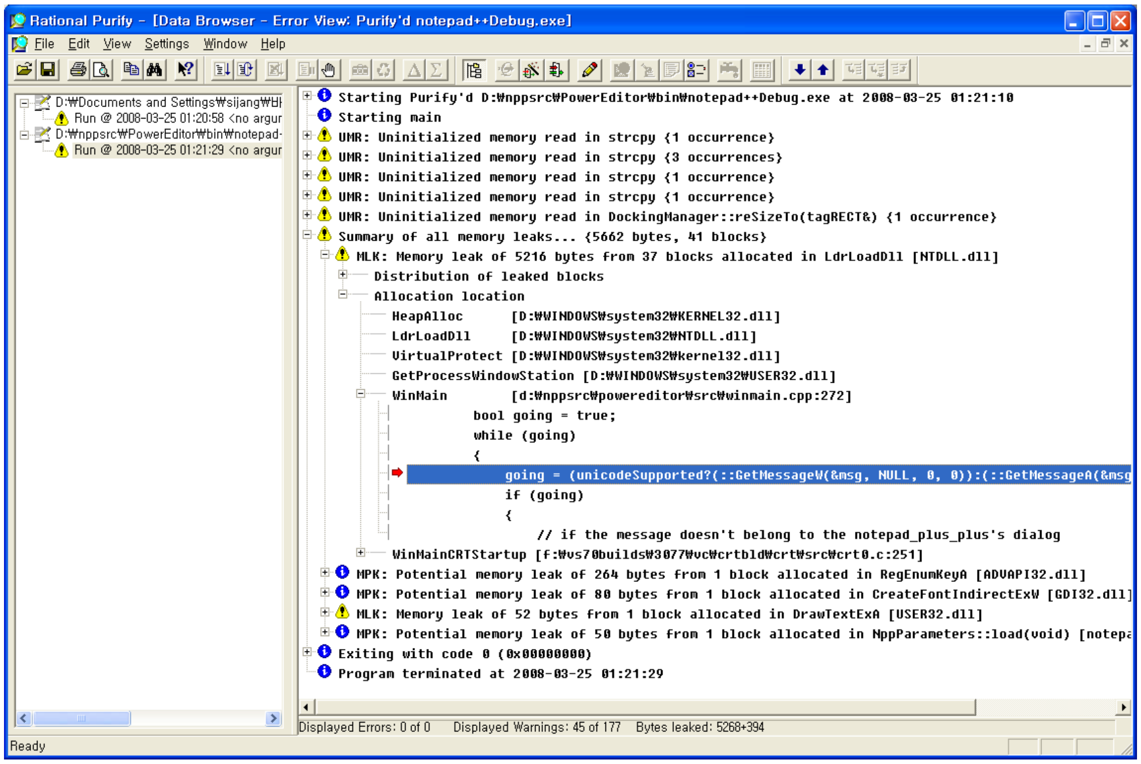
```
Starting Purify'd D:₩Program Files₩Notepad++₩notepad++.exe at 2008-03-25 00:29:40
Starting main
PAR: Insufficient parameters match format string at (0x105a40) {1 occurrence}
Summary of all memory leaks... {15425 bytes, 58 blocks}
  MPK: Potential memory leak of 6224 bytes from 4 blocks allocated in CreateWindowExW [USER32.dll]
  MLK: Memory leak of 5216 bytes from 37 blocks allocated in LdrLoadDll [NTDLL.dll]
    Distribution of leaked blocks
    Allocation location
      HeapAlloc      [D:₩WINDOWS₩system32₩KERNEL32.dll]
      LdrLoadDll     [D:₩WINDOWS₩system32₩NTDLL.dll]
      VirtualProtect [D:₩WINDOWS₩system32₩kernel32.dll]
      GetProcessWindowStation [D:₩WINDOWS₩system32₩USER32.dll]
  MLK: Memory leak of 1768 bytes from 3 blocks allocated in CreateWindowExW [USER32.dll]
    Distribution of leaked blocks
    Allocation location
      LocalAlloc     [D:₩WINDOWS₩system32₩KERNEL32.dll]
      CreateWindowExW [D:₩WINDOWS₩system32₩USER32.dll]
  MLK: Memory leak of 1019 bytes from 4 blocks allocated in LdrLoadDll [NTDLL.dll]
  MLK: Memory leak of 297 bytes from 1 block allocated in GetProcessHeap [kernel32.dll]
  MLK: Memory leak of 297 bytes from 1 block allocated in IsValidIid [OLE32.dll]
  MPK: Potential memory leak of 264 bytes from 1 block allocated in RegEnumKeyA [ADVAPI32.dll]
  MLK: Memory leak of 84 bytes from 1 block allocated in LdrLoadDll [NTDLL.dll]
  MLK: Memory leak of 68 bytes from 2 blocks allocated in ScrollDC [USER32.dll]
  MLK: Memory leak of 64 bytes from 1 block allocated in CreateWindowExW [USER32.dll]
  MLK: Memory leak of 52 bytes from 1 block allocated in DrawTextExA [USER32.dll]
  MPK: Potential memory leak of 50 bytes from 1 block allocated in LdrLoadDll [NTDLL.dll]
  MPK: Potential memory leak of 22 bytes from 1 block allocated in LdrLoadDll [NTDLL.dll]
FIM: Freeing invalid memory in LocalFree {36 occurrences}
Exiting with code 0 (0x00000000)
Program terminated at 2008-03-25 00:30:22

Displayed Errors: 36 of 36    Displayed Warnings: 52 of 227    Bytes leaked: 8865+6560
```

Figure 3 A Purify main window (without debug information)

To find defects of Notepad++, we downloaded the source code of Purify from the Internet. According to the developers of Notepad++, we can compile the source code with MS Visual C++ 7.0 or MinGW. Therefore, we chose MS Visual C++ 7.0 as a compiler.

If we use a test program which contains debug information, we can track the exact point of source code in the test program easily. Figure 4 shows an example of error list. Purify provides not only which code includes defects, but also the sequence of function call. When we clicked a source code in the right tree view, Purify showed the actual source code with MS Visual C++. And we could track and review the source code easily. In this mini project, we evaluated Purify with debug information.


Rational Purify - [Data Browser - Error View: Purify'd notepad++Debug.exe]
File Edit View Settings Window Help
D:₩Documents and Settings₩sijang₩바
Run @ 2008-03-25 01:20:58 <no argur
D:₩nppsrc₩PowerEditor₩bin₩notepad-
Run @ 2008-03-25 01:21:29 <no argur

```
Starting Purify'd D:₩nppsrc₩PowerEditor₩bin₩notepad++Debug.exe at 2008-03-25 01:21:10
Starting main
UMR: Uninitialized memory read in strcpy {1 occurrence}
UMR: Uninitialized memory read in strcpy {3 occurrences}
UMR: Uninitialized memory read in strcpy {1 occurrence}
UMR: Uninitialized memory read in strcpy {1 occurrence}
UMR: Uninitialized memory read in DockingManager::reSizeTo(tagRECT&) {1 occurrence}
Summary of all memory leaks... {5662 bytes, 41 blocks}
  MLK: Memory leak of 5216 bytes from 37 blocks allocated in LdrLoadDll [NTDLL.dll]
    Distribution of leaked blocks
    Allocation location
      HeapAlloc      [D:₩WINDOWS₩system32₩KERNEL32.dll]
      LdrLoadDll     [D:₩WINDOWS₩system32₩NTDLL.dll]
      VirtualProtect [D:₩WINDOWS₩system32₩kernel32.dll]
      GetProcessWindowStation [D:₩WINDOWS₩system32₩USER32.dll]
      WinMain        [d:₩nppsrc₩powereditor₩src₩winmain.cpp:272]
                bool going = true;
                while (going)
                {
                    going = (unicodeSupported?(::GetMessageW(&msg, NULL, 0, 0)):(::GetMessageA(&msg
                    if (going)
                    {
                        // if the message doesn't belong to the notepad_plus_plus's dialog
      WinMainCRTStartup [f:₩vs70builds₩3077₩vc₩crtbld₩crt₩src₩crt0.c:251]
  MPK: Potential memory leak of 264 bytes from 1 block allocated in RegEnumKeyA [ADVAPI32.dll]
  MPK: Potential memory leak of 80 bytes from 1 block allocated in CreateFontIndirectExW [GDI32.dll]
  MLK: Memory leak of 52 bytes from 1 block allocated in DrawTextExA [USER32.dll]
  MPK: Potential memory leak of 50 bytes from 1 block allocated in NppParameters::load(void) [notepa
Exiting with code 0 (0x00000000)
Program terminated at 2008-03-25 01:21:29
```

Displayed Errors: 0 of 0   Displayed Warnings: 45 of 177   Bytes leaked: 5268+394

Ready

Figure 4 A Purify main window (with debug information)

To evaluate the soundness of Purify, we categorized defects into four classes: true-positive, false-positive, false-negative, and others.

- True-positive: represents that Purify notifies real defects.
- False-positive: represents that Purify notifies non-defects.
- False-negative: represents that Purify does not notifies real defects.
- Others: represent notifications which do not belong to above three classes.

3.3 The defect-seeded program

The purpose of creating a defect-seeded program is to compare the tool with human inspection and to find false-negatives.

We decided to compare human inspection with the usage of the tool in order to evaluate the efficiency of using the tool. Finding defects using a computer – Purify in this case – would be faster than without it. Our concerns are how much faster.

One of the team members developed a simple sorted linked list code, and then he seeded several defects. The number of seeded defects was not told to the other three members. The three members are supposed to find the seeded defects within 10 minutes.

One of the team members used Purify to debug the buggy code. Elapsed time to fix all reported defects (not all seeded defects) was recorded.

The percentage of false-negatives and false-positives was also recorded.

The simple buggy code that is used for group inspection and debugging is as follows:

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

#define MAX_ELEMENT_LENGTH 10
typedef struct NODE
{
    NODE * next;
    char element[MAX_ELEMENT_LENGTH];
}   tNode;

NODE * newList();
void insert( NODE * header, char * element);
int find( NODE * header, char * element);
char * getInput( char * string);
void copy(char *target, char *source, int n);
void printAll( NODE * header);

int main( int argc, char *argv[])
{
    char *test=(char*)malloc(10); // alloc #1
    char c = '\"';
    int len;
    NODE * header;
    header = newList();

    test = getInput("element"); // error #1
    len = strlen(test); // error #2
    while (len>1)
    {
        test[len-1] = 0;
        insert(header, test);
        test = getInput(&c); // error #3
        len = strlen(test);
    }

    printf("Final list is as follows...\n");
    printAll(header);

    if ( find(header, "aob"))
        printf("aob Found!!\n");
    else
        printf("no aob..\n");
    // error #4: no free functions
}

NODE * newList()
{
    NODE * newNode;
    newNode->next = NULL; // error #5
    return newNode;
}

void insert( NODE * header, char * element)
{
    NODE * current = header;
    while ( current->next)
    {
            if ( strcmp( element, current->next->element)<0)
            {
                NODE * node;
                node->next = current->next; // error #6
                current->next = node;
                int len = strlen(element);
                copy( node->element, element, len);
                return;
            }
            current=current->next;
        }
        NODE * node;
        current->next = node; // error #7
        node->next = NULL; // error #8
        int len = strlen(element);
        copy( node->element, element, len);
        return;
    }

    int find( NODE * header, char * element)
    {
        NODE * current = header;
        while ( current->next)
        {
            if ( strcmp(current->element, element)==0) // error #9
                return 1;
            current = current->next;
        }
        return 0;
    }

    char * getInput( char * string)
    {
        char * input = (char *)malloc(80); // alloc #2
        printf("Input %s(exit:just press enter key): ", string);
        fgets(input,80,stdin); // error #10
        return input;
    }

    void copy( char *target, char *source, int n)
    {
        strncpy( target, source, n); // error #11
        target[n] = 0;
        return;
    }

    void printAll( NODE * header)
    {
        NODE * current = header;
        printf("=== START ===\n");
        while ( current->next)
        {
            current = current->next;
            printf("%s\n", current->element);
        }
        printf("=== END ===\n");
    }
```

Figure 5 The buggy source code

The program simply inserts elements into sorted linked list, and then finds whether the list contains element ‘aob’ or not. There are nine errors injected intentionally.

Note that the code is written with Visual C++ 2008 Express Edition and compiled in Debug mode in order to give call stack information to Purify as explained earlier.

4 Evaluation Results

4.1 Overall Evaluation

Questionnaire	Average score
Installation easiness (1: Poor ~ 5: Excellent)	2.50
Overall Interface Intuitiveness (1: Poor ~ 5: Excellent)	3.25
Easy to learn how to use (1: Poor ~ 5: Excellent)	3.75
Easy to understand the error summary (1: Poor ~ 5: Excellent)	3.50
Easy to find the location of errors (1: Poor ~ 5: Excellent)	4.00
Easy to follow the execution trace of Source Code (1: Poor ~ 5: Excellent)	3.75

Figure 6 Tool usability survey result

According to the Figure 6, our members experienced difficulty to install Purify. Overall interface was properly intuitive to the members, and understandability to error report (summary) was also close to standard (i.e. 3). Learnability and execution trace of source code are relatively better. Finally, our members thought it is very effective to find the location of defects in source code. To sum up, our members satisfied about overall usability but thought it is difficult to install.

We firstly contrasted between the size of original program and program that is executed by Purify in order to measure performance (memory usage and latency), and then we checked the time to close 100 documents after the Notepad++ had already opened 100 documents. As a result, our defect-seeded program consumed 10 times of memory (1MB by original and 10MB by Purify execution) and Notepad++ also consumed about 5 times of memory (9MB vs. 50MB). For latency, we could measure 1.2 seconds with original program to close 100 documents but it took 9.4 seconds to close 100 documents with the program executed by Purify. It is hard to generalize the performance of Purify, but it is sure that Purify affects significantly for memory usage and latency.

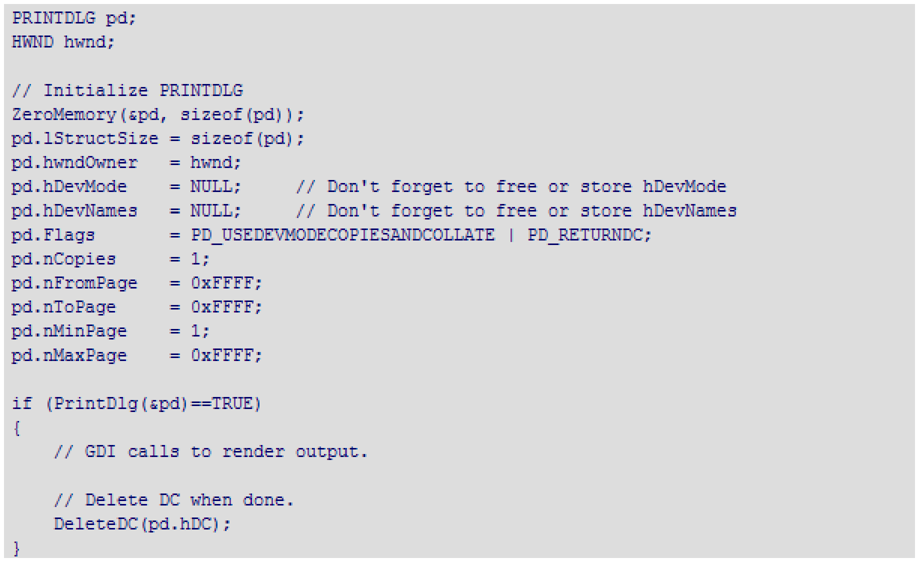
4.2 The Notepad++

As we mentioned in the previous chapter, we categorized defects into four classes.

4.2.1 True-positive

True-positive represents that Purify notifies errors which are real defects. With Purify, we found several true-positive errors from the Notepad++ source code.

Figure 7 shows an example of `PrintDlg()` usage. According to the MSDN specification, users have to free or store the values of `hDevMode` and `hDevNames` which are parts of `PRINTDLG`. The developers of the Notepad++ did not insert a memory free instruction, and Purify found this defects. After inserting the memory free code, Purify did not notify that anymore.



```
PRINTDLG pd;
HWND hwnd;

// Initialize PRINTDLG
ZeroMemory(&pd, sizeof(pd));
pd.lStructSize = sizeof(pd);
pd.hwndOwner   = hwnd;
pd.hDevMode    = NULL;     // Don't forget to free or store hDevMode
pd.hDevNames   = NULL;     // Don't forget to free or store hDevNames
pd.Flags       = PD_USEDEVMODECOPIESANDCOLLATE | PD_RETURNDC;
pd.nCopies     = 1;
pd.nFromPage   = 0xFFFF;
pd.nToPage     = 0xFFFF;
pd.nMinPage    = 1;
pd.nMaxPage    = 0xFFFF;

if (PrintDlg(&pd)==TRUE)
{
    // GDI calls to render output.

    // Delete DC when done.
    DeleteDC(pd.hDC);
}
```

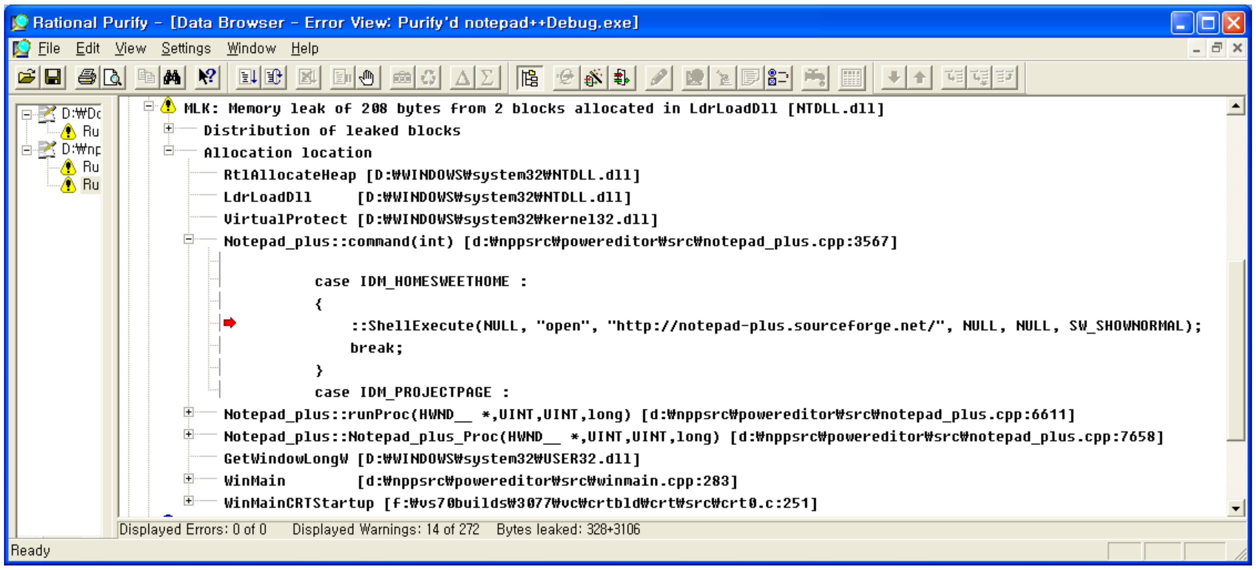
Figure 7 An example of PrintDlg() usage

In another class, the developers used `SHGetSpecialFolderLocation()` function. According to the MSDN specification, users have a responsibility for freeing a memory block with the `CoTaskMemFree()` function. However, the developers did not free the memory, and Purify notify this defect. This defect also removed by inserting a memory free code.

And Purify found many minor defects related with memory initialization. In many cases, for example, the developers did not initialize some variables, Purify found these defects.

4.2.2 False-positive

In most cases, Purify found defects correctly, but sometimes it notified false-positive errors. Figure 8 shows an example of false-positive. As you can see, Purify reported an error which uses `ShellExecute()` function which is provided by the operating system. Because the parameters of the functions are just constants, there cannot be memory leak obviously. There are two possible scenarios. One is that Microsoft provides buggy DLL files, and the other one is that Purify has some defects.

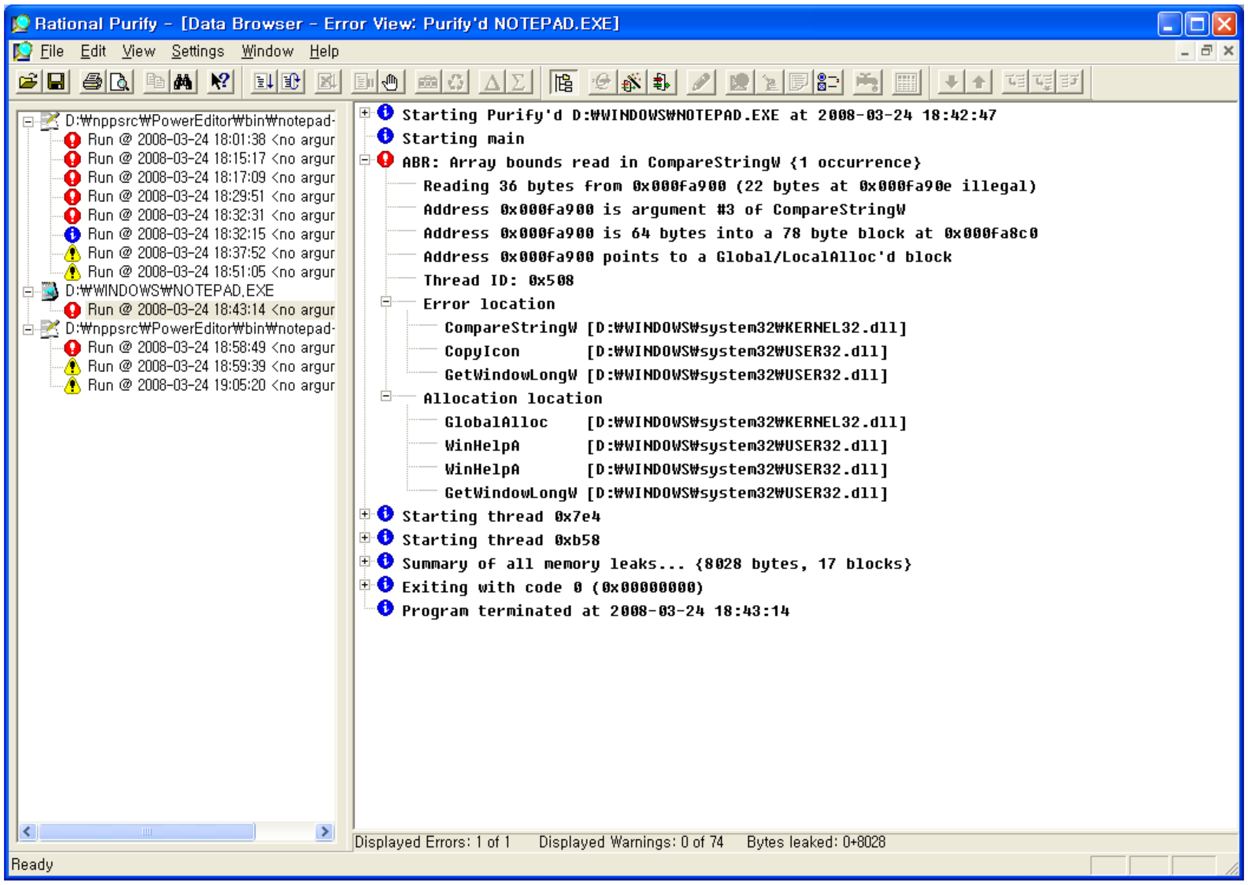

Rational Purify - [Data Browser - Error View: Purify'd notepad++Debug.exe]
File Edit View Settings Window Help
D:₩Dc
Ru
D:₩np
Ru
Ru
Ready

```
MLK: Memory leak of 208 bytes from 2 blocks allocated in LdrLoadDll [NTDLL.dll]
  Distribution of leaked blocks
  Allocation location
    RtlAllocateHeap [D:₩WINDOWS₩system32₩NTDLL.dll]
    LdrLoadDll      [D:₩WINDOWS₩system32₩NTDLL.dll]
    VirtualProtect [D:₩WINDOWS₩system32₩kernel32.dll]
    Notepad_plus::command(int) [d:₩nppsrc₩powereditor₩src₩notepad_plus.cpp:3567]

            case IDM_HOMESWEETHOME :
            {
                ::ShellExecute(NULL, "open", "http://notepad-plus.sourceforge.net/", NULL, NULL, SW_SHOWNORMAL);
                break;
            }
            case IDM_PROJECTPAGE :
    Notepad_plus::runProc(HWND__ *,UINT,UINT,long) [d:₩nppsrc₩powereditor₩src₩notepad_plus.cpp:6611]
    Notepad_plus::Notepad_plus_Proc(HWND__ *,UINT,UINT,long) [d:₩nppsrc₩powereditor₩src₩notepad_plus.cpp:7658]
    GetWindowLongW [D:₩WINDOWS₩system32₩USER32.dll]
    WinMain        [d:₩nppsrc₩powereditor₩src₩winmain.cpp:283]
    WinMainCRTStartup [f:₩vs70builds₩3077₩vc₩crtbld₩crt₩src₩crt0.c:251]
Displayed Errors: 0 of 0    Displayed Warnings: 14 of 272    Bytes leaked: 328+3106
```

Figure 8 An example of false-positive

Purify notified a defect with `PrintDlg()` function which is provided by Microsoft, but we could not find any wrong codes in the source code. We thought that it also a false-positive error. To clarify this, we experimented with MS Notepad which is very simple editor. When we open a print dialog, Purify notified an error which is same with Notepad++. Figure 9 shows the result of this test. However, we cannot sure it is a defect of Microsoft Notepad or a defect of Purify because we cannot access the source codes of them.


Rational Purify - [Data Browser - Error View: Purify'd NOTEPAD.EXE]
File Edit View Settings Window Help
Ready

```
D:₩nppsrc₩PowerEditor₩bin₩notepad-
    Run @ 2008-03-24 18:01:38 <no argur
    Run @ 2008-03-24 18:15:17 <no argur
    Run @ 2008-03-24 18:17:09 <no argur
    Run @ 2008-03-24 18:29:51 <no argur
    Run @ 2008-03-24 18:32:31 <no argur
    Run @ 2008-03-24 18:32:15 <no argur
    Run @ 2008-03-24 18:37:52 <no argur
    Run @ 2008-03-24 18:51:05 <no argur
D:₩WINDOWS₩NOTEPAD.EXE
    Run @ 2008-03-24 18:43:14 <no argur
D:₩nppsrc₩PowerEditor₩bin₩notepad-
    Run @ 2008-03-24 18:58:49 <no argur
    Run @ 2008-03-24 18:59:39 <no argur
    Run @ 2008-03-24 19:05:20 <no argur
```

```
Starting Purify'd D:₩WINDOWS₩NOTEPAD.EXE at 2008-03-24 18:42:47
Starting main
ABR: Array bounds read in CompareStringW {1 occurrence}
   Reading 36 bytes from 0x000fa900 (22 bytes at 0x000fa90e illegal)
   Address 0x000fa900 is argument #3 of CompareStringW
   Address 0x000fa900 is 64 bytes into a 78 byte block at 0x000fa8c0
   Address 0x000fa900 points to a Global/LocalAlloc'd block
   Thread ID: 0x508
   Error location
      CompareStringW [D:₩WINDOWS₩system32₩KERNEL32.dll]
      CopyIcon       [D:₩WINDOWS₩system32₩USER32.dll]
      GetWindowLongW [D:₩WINDOWS₩system32₩USER32.dll]
   Allocation location
      GlobalAlloc    [D:₩WINDOWS₩system32₩KERNEL32.dll]
      WinHelpA       [D:₩WINDOWS₩system32₩USER32.dll]
      WinHelpA       [D:₩WINDOWS₩system32₩USER32.dll]
      GetWindowLongW [D:₩WINDOWS₩system32₩USER32.dll]
Starting thread 0x7e4
Starting thread 0xb58
Summary of all memory leaks... {8028 bytes, 17 blocks}
Exiting with code 0 (0x00000000)
Program terminated at 2008-03-24 18:43:14
```

```
Displayed Errors: 1 of 1    Displayed Warnings: 0 of 74    Bytes leaked: 0+8028
```

Figure 9 A test with Microsoft Notepad

4.2.3 False-negative

We found a critical error which can be a cause of buffer overrun. This error was discovered by inspection of team members, but Purify did not notify this error. Figure 10 shows an example of false-negative. As you can see, the developers used `_itoa()` function which converts an integer to a string, but the developers assigned only four-byte long array. According to the MSDN specification, users have to assign long enough array when using `_itoa()` function. This code might make some problems because of buffer overruns in stack.

```
char * ScintillaEditView::attatchDefaultDoc(int nb)
{
    char title[10];
    char nb_str[4];

    strcat(strcpy(title, UNTITLED_STR), _itoa(nb, nb_str, 10));

    // get the doc pointer attached (by default) on the view Scintilla
    Document doc = execute(SCI_GETDOCPOINTER, 0, 0);

    // create the entry for our list
```

```
        _buffers.push_back(Buffer(doc, title));

        // set current index to 0
        _currentIndex = 0;

        return _buffers[_currentIndex]._fullPathName;
    }
```

Figure 10 An example of false-negative

To test Purify, we reduced the size of `nb_str` by 2, and executed the Notepad++. Because the code is related with generating new documents, we generated more than 1,000 documents. Figure 11 shows a result of this test. Purify gives sequential number to new documents, but after generating 999 documents, Purify gives wrong numbers, as you can see in Figure 11.

Even though buffer overrun is a kind of critical problem, Purify did not notify this error.

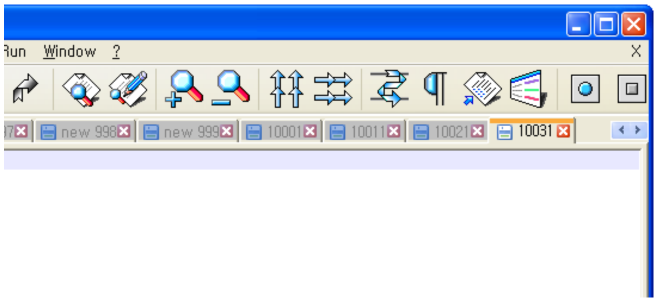


Figure 11 An example of false-negative test

4.2.4 Others

By using Purify, we found that the developers used some strange codes. Purify notified this defect as an uninitialized memory usage. Figure 12 shows an example of strange usages. The type of the first parameter of `append()` function is pointer of character array, and the type of second parameter is integer. Instead of using an array, the developers used just a pointer of character. Even though this code does not make a problem, we think that this kind of usage is an abnormal usage. To soundness of the program, this kind of usage should be removed.

```
        …
        char realc = (char) c;
        outString->append( &realc, 1 );
        ++i;
        …
```

Figure 12 An example of strange usage

4.3 The defect-seeded program

4.3.1 Inspection

During inspection, three members could find 7 defects out of 11 defects. Error #1, #4, #9 and #11 were not identified by group inspection.

Team members tend to miss memory error that violation does not occur close to allocation; error #1 allocates memory in separate function.

Regarding error #4, no member could find any memory leak despite that the code does not contain any `free` function calls. The result was surprising because all members were well aware that the purpose of the inspection was to find memory defects.

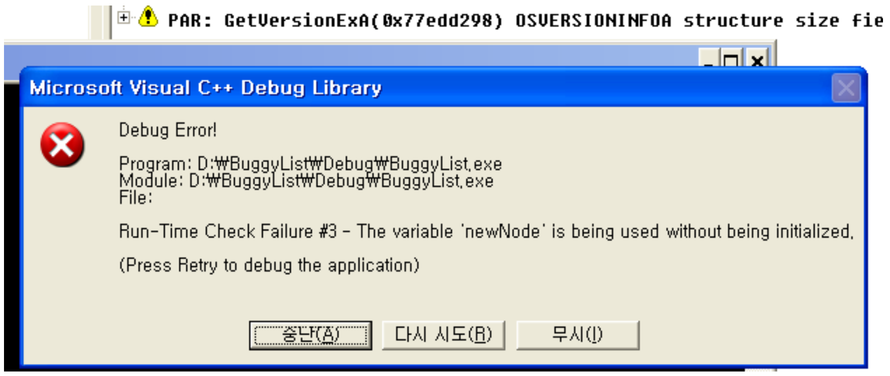
Error #9 is hard to identify without simulation.

Error #11 is also hard to find for humans because the size of target is defined within `NODE` structure which is 10, and the length of source is defined in run-time which can be up to 80 bytes including null terminator.

4.3.2 Debugging with Purify

Error #5, #6, #7, and #8

After executing the initial buggy code, we got a critical error message from Windows XP operating system in Figure 13.



PAR: GetVersionExA(0x77edd298) OSVERSIONINFOA structure size fie

Microsoft Visual C++ Debug Library

Debug Error!

Program: D:₩BuggyList₩Debug₩BuggyList.exe
Module: D:₩BuggyList₩Debug₩BuggyList.exe
File:

Run-Time Check Failure #3 - The variable 'newNode' is being used without being initialized.

(Press Retry to debug the application)

Figure 13 Uninitialized variable error

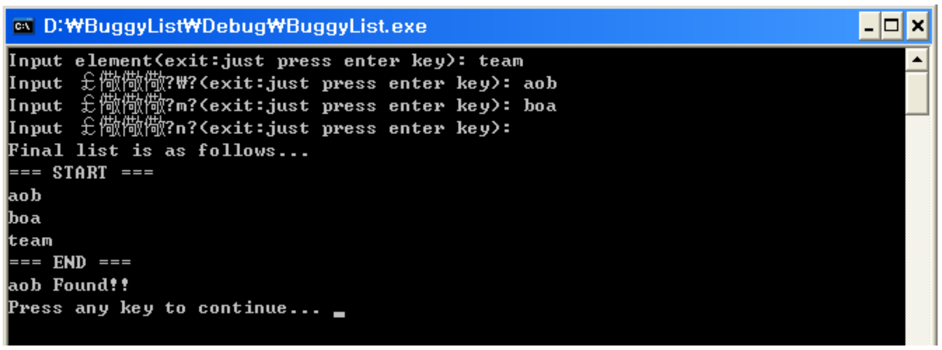
We fixed error #5, #6, #7, and #8 in the previous code. Three of them (#5, #6, and #8) are Uninitialized Memory Read (UMR) error [1], whereas the error #7 is Uninitialized Memory Copy (UMC) error [1]. These errors normally cause abnormal program halt, so it will be found while simply running the program if the binary is compiled in Debug mode. We fixed all four defects by simply adding memory allocation calls as follows:

```
NODE * newNode;
newNode = (NODE*)malloc(sizeof(NODE));
```

Error #3 (Not detected)

We input three elements including “team,” “aob,” and “boa.” The following screenshot shows something wrong with the program because sentences from second occurrence are not clearly written in screen.

This is because of the error #3. However, Purify cannot find nor warn the error. The variable `c` is defined as a single character. And `getInput` uses address of `c` as a pointer to characters despite there is no guarantee of following null terminator. This is buffer overruns in local stack.

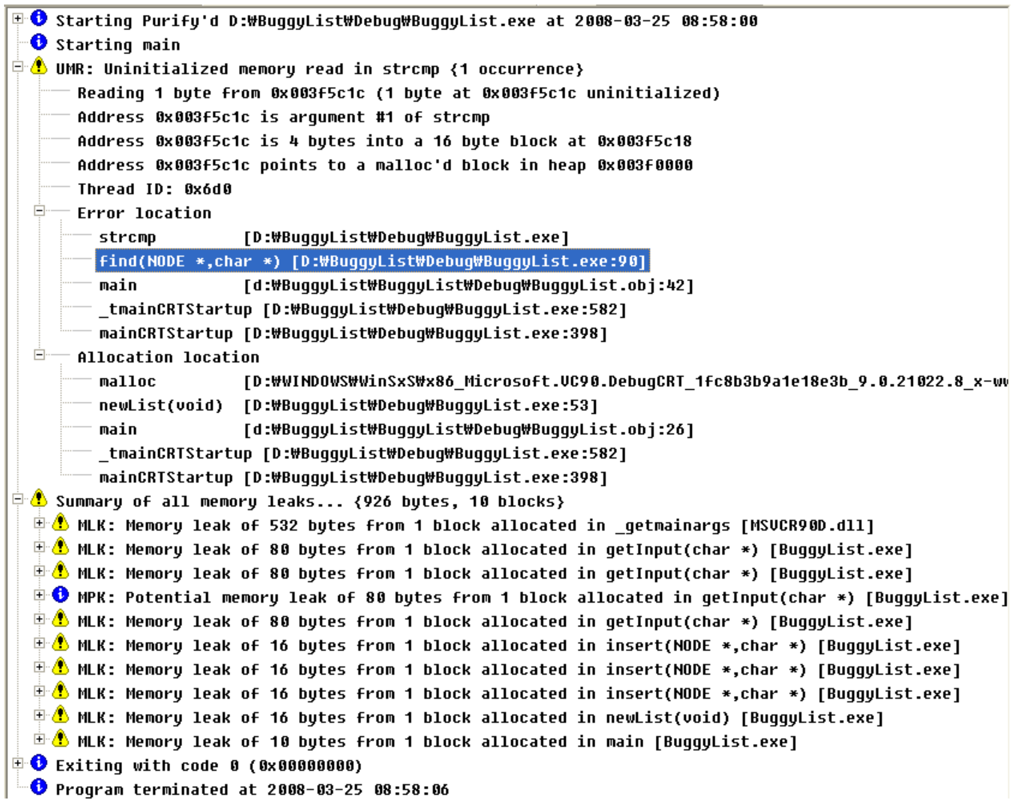


```
D:₩BuggyList₩Debug₩BuggyList.exe
Input element(exit:just press enter key): team
Input ￡儆儆儆?₩?(exit:just press enter key): aob
Input ￡儆儆儆?m?(exit:just press enter key): boa
Input ￡儆儆儆?n?(exit:just press enter key):
Final list is as follows...
=== START ===
aob
boa
team
=== END ===
aob Found!!
Press any key to continue...
```

Figure 14 Simple execution

Error #9 (Detected)

Following screenshot is the result of Purify with the buggy program. The UMR warning is `strcmp` in 88th line and matches to the error #9.



```
Starting Purify'd D:₩BuggyList₩Debug₩BuggyList.exe at 2008-03-25 08:58:00
Starting main
UMR: Uninitialized memory read in strcmp {1 occurrence}
    Reading 1 byte from 0x003f5c1c (1 byte at 0x003f5c1c uninitialized)
    Address 0x003f5c1c is argument #1 of strcmp
    Address 0x003f5c1c is 4 bytes into a 16 byte block at 0x003f5c18
    Address 0x003f5c1c points to a malloc'd block in heap 0x003f0000
    Thread ID: 0x6d0
    Error location
        strcmp          [D:₩BuggyList₩Debug₩BuggyList.exe]
        find(NODE *,char *) [D:₩BuggyList₩Debug₩BuggyList.exe:90]
        main            [d:₩BuggyList₩BuggyList₩Debug₩BuggyList.obj:42]
        _tmainCRTStartup [D:₩BuggyList₩Debug₩BuggyList.exe:582]
        mainCRTStartup  [D:₩BuggyList₩Debug₩BuggyList.exe:398]
    Allocation location
        malloc          [D:₩WINDOWS₩WinSxS₩x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_9.0.21022.8_x-w
        newList(void)   [D:₩BuggyList₩Debug₩BuggyList.exe:53]
        main            [d:₩BuggyList₩BuggyList₩Debug₩BuggyList.obj:26]
        _tmainCRTStartup [D:₩BuggyList₩Debug₩BuggyList.exe:582]
        mainCRTStartup  [D:₩BuggyList₩Debug₩BuggyList.exe:398]
Summary of all memory leaks... {926 bytes, 10 blocks}
    MLK: Memory leak of 532 bytes from 1 block allocated in _getmainargs [MSVCR90D.dll]
    MLK: Memory leak of 80 bytes from 1 block allocated in getInput(char *) [BuggyList.exe]
    MLK: Memory leak of 80 bytes from 1 block allocated in getInput(char *) [BuggyList.exe]
    MPK: Potential memory leak of 80 bytes from 1 block allocated in getInput(char *) [BuggyList.exe]
    MLK: Memory leak of 80 bytes from 1 block allocated in getInput(char *) [BuggyList.exe]
    MLK: Memory leak of 16 bytes from 1 block allocated in insert(NODE *,char *) [BuggyList.exe]
    MLK: Memory leak of 16 bytes from 1 block allocated in insert(NODE *,char *) [BuggyList.exe]
    MLK: Memory leak of 16 bytes from 1 block allocated in insert(NODE *,char *) [BuggyList.exe]
    MLK: Memory leak of 16 bytes from 1 block allocated in newList(void) [BuggyList.exe]
    MLK: Memory leak of 10 bytes from 1 block allocated in main [BuggyList.exe]
Exiting with code 0 (0x00000000)
Program terminated at 2008-03-25 08:58:06
```

Figure 15 Result screen of Purify

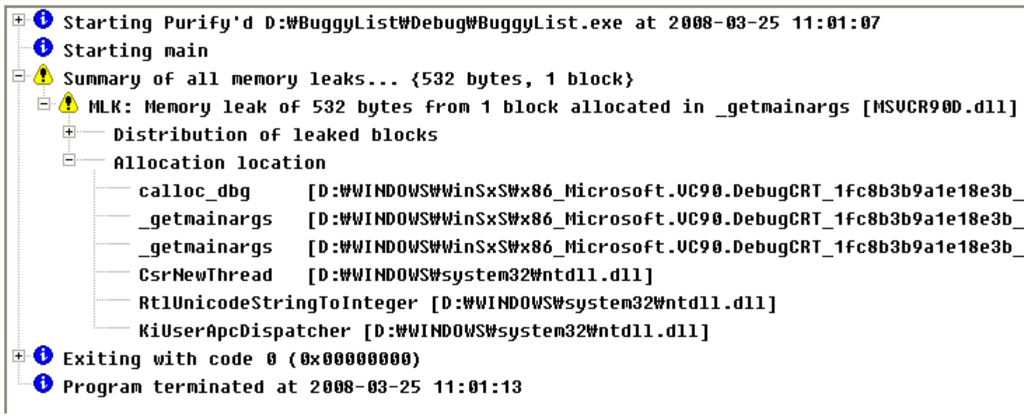
The first `current` in loop is `header` at first time, so `current->element` will be definitely uninitialized in this function. This is UMR, too. No member could find this error because it was very hard to find without simulation. We modified the code so that the first `current` points out the next node to the `header`.

Error #1 and #4 (Detected)

There are 10 Memory Leaks (MLKs) in figure 3. These memory leaks were not identified by inspection, but Purify successfully identified them plus one strange reporting. Information about the strange 532 bytes of memory leak report can be referred to Appendix. (This is the second false negative error.)

We entered four inputs including the last blank input, and Purify showed three MLKs and one MPK. Four 16 bytes leaks represent three elements and one header node. The final 10 bytes leak corresponds to `alloc #1` in source code.

We added free function calls. No more memory leaks related to our code were found as Figure 16 shows.



```
Starting Purify'd D:\BuggyList\Debug\BuggyList.exe at 2008-03-25 11:01:07
Starting main
Summary of all memory leaks... {532 bytes, 1 block}
  MLK: Memory leak of 532 bytes from 1 block allocated in _getmainargs [MSVCR90D.dll]
    Distribution of leaked blocks
    Allocation location
      calloc_dbg     [D:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_
      _getmainargs   [D:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_
      _getmainargs   [D:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_
      CsrNewThread   [D:\WINDOWS\system32\ntdll.dll]
      RtlUnicodeStringToInteger [D:\WINDOWS\system32\ntdll.dll]
      KiUserApcDispatcher [D:\WINDOWS\system32\ntdll.dll]
Exiting with code 0 (0x00000000)
Program terminated at 2008-03-25 11:01:13
```

Figure 16 Final result of Purify with the buggy code

Error #2 and #10 (Not detected)

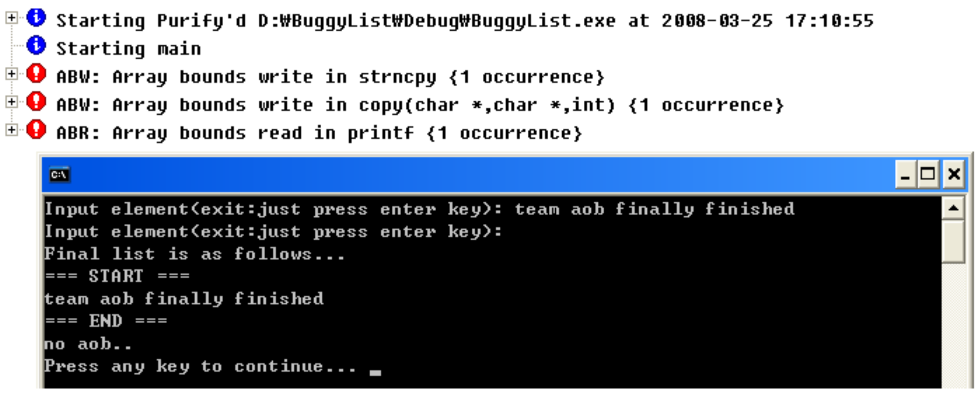
When `malloc` returns `NULL` due to some external reasons, null dereference occurs at error #2 and #10.

These two errors are impossible to identify unless there is `malloc` stubs implemented that returns NULL or `malloc` returns `NULL` due to some reasons like not enough memory situation. This is limitation of dynamic analysis methods.

Error #11 (Not detected, but possible to detect with different inputs)

The size of `element` is 10 bytes. However, more than 10 bytes can be written to `element` according to the line of error #11 because the length of source string can be up to 80 bytes including null terminator.

So, we tested with longer input to check whether Purify identifies it or not. It successfully identified Array out of bounds error as Figure 17 shows.



```
Starting Purify'd D:₩BuggyList₩Debug₩BuggyList.exe at 2008-03-25 17:10:55
Starting main
ABW: Array bounds write in strncpy {1 occurrence}
ABW: Array bounds write in copy(char *,char *,int) {1 occurrence}
ABR: Array bounds read in printf {1 occurrence}
```

```
Input element(exit:just press enter key): team aob finally finished
Input element(exit:just press enter key):
Final list is as follows...
=== START ===
team aob finally finished
=== END ===
no aob..
Press any key to continue...
```

Figure 17 Successful error detection with longer input

5 Conclusion

5.1 Benefits

The most important benefit of using Rational Purify is on the types of errors Purify detects because typically run-time memory errors and warnings are very difficult errors to locate. The symptoms of incorrect memory usage are unpredictable and usually appear far from the cause of the error. An inspection on code might be hard to detect these types of errors but Rational Purify can effectively detect them. With a program compiled in debug mode, Purify can directly point out the error coded line, which is very helpful for developers to correct the code.

5.2 Drawbacks

Although Purify has strong benefits, it has some drawbacks. Because Purify focuses on run-time error detection, it is natural that it has unsoundness in testing; it is almost impossible to have 100% coverage of code and even its analysis has some false negative reports of errors, which is illustrated in evaluation results section.

In addition, if a tested program stops by a fatal error during the testing, then Purify cannot continue the testing because Purify can only test a running program; this makes testing jobs more tedious. In case of using a static analysis tool, we might test the whole code at one time, which might be not practical in real, but with Purify we have to correct the error in advance to continue on the testing whenever we meet that kind of fatal errors.

5.3 Scope of Applicability

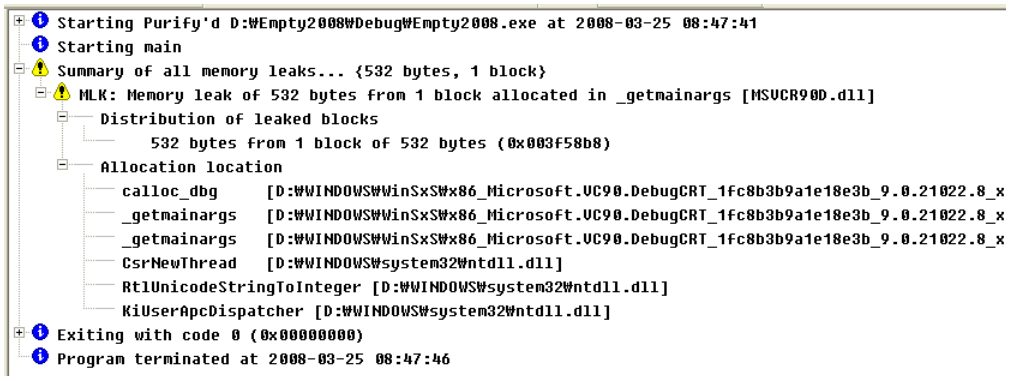
Purify might be most applicable to the systems to which memory management is critical, for example, embedded system domain: because typical embedded systems have very limited memory resource, the misuse of memory might be critical to the systems. In addition, unlikely with static analysis tools, since Purify cannot continue to test when the tested program stop, it might be useful to apply Purify to comparatively stable code during the development phase or testing phase.

Purify has also other capability for java language: it can evaluate the CPU and memory usage of program. Since the system we develop in our studio project will be develop with java, Purify can be very useful for us to develop quality system.

A. Appendix

Purify identified 532 bytes of Memory Leak in following simple code compiled in Visual C++ 2008 Express Edition as Figure 18 denotes. The reason could be one of the followings: tool reported false positive or definite memory leak is being created by Visual C++ 2008 Express Edition.

```
int main()
{
    return 0;
}
```



```
⊞ Starting Purify'd D:₩Empty2008₩Debug₩Empty2008.exe at 2008-03-25 08:47:41
  Starting main
⊟ Summary of all memory leaks... {532 bytes, 1 block}
   ⊟ MLK: Memory leak of 532 bytes from 1 block allocated in _getmainargs [MSVCR90D.dll]
      ⊟ Distribution of leaked blocks
              532 bytes from 1 block of 532 bytes (0x003f58b8)
      ⊟ Allocation location
           calloc_dbg     [D:₩WINDOWS₩WinSxS₩x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_9.0.21022.8_x
           _getmainargs   [D:₩WINDOWS₩WinSxS₩x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_9.0.21022.8_x
           _getmainargs   [D:₩WINDOWS₩WinSxS₩x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3b_9.0.21022.8_x
           CsrNewThread   [D:₩WINDOWS₩system32₩ntdll.dll]
           RtlUnicodeStringToInteger [D:₩WINDOWS₩system32₩ntdll.dll]
           KiUserApcDispatcher [D:₩WINDOWS₩system32₩ntdll.dll]
⊞ Exiting with code 0 (0x00000000)
  Program terminated at 2008-03-25 08:47:46
```

Figure 18 Result screen of Empty project in Purify

The code is compiled in debug mode, so Uninitialized Memory Read (UMR) is notified in Microsoft Visual C++ Debug Library alert window whenever there is a serious memory access violation.

B. References

[1] http://www.ibm.com/developerworks/rational/library/06/0822_satish-giridhar/
[2] http://notepad-plus.sourceforge.net
[3] http://publibfp.boulder.ibm.com/epubs/pdf/12653120.pdf