n.bug Documentation

Program Version 1.0.0.15 RC2, Documentation Version 1.0

Introduction

n.bug is a library call trace implementation for Windows NT operating systems. It is meant as a tool to quickly analyse specific library calls a binary program makes during runtime.

A typical n.bug trace run consists of the following steps:

1. User selects the functions to trace
2. User selects either a running process or a program path (called the *target*)
3. User initiates trace
4. n.bug attaches a debugger engine to the target and executes it or resumes execution
5. (optional) User interacts with target
6. User terminates trace
7. User inspects trace results and optionally saves them to a file
8. (optional) User repeats the process with different settings

As the result of a trace session, n.bug outputs the following information for each library function call detected and matching the trace definition:

- The caller's return address (points to the instruction after the call instruction)
- The module and function name called
- The type, name and value of the parameter upon entry into the library function
- The type, name and value of the parameter upon exit from the library function
- The return value of the call

The results of a trace session can be saved into a text file for future reference and documentation purposes.

Runtime analysis theory

For readers not familiar with the theory behind runtime analysis, this section will give an overview on what techniques are used in n.bug. Readers already familiar with the background or comparable tools on other operating system platforms (such as ltrace on Linux) can safely skip ahead.

General concept

Runtime analysis inspects the code of a target process during its execution. In contrast to static analysis, which takes the processes code and data information and tries to determine the possible execution paths, runtime analysis does not suffer from the many cases in which static analysis cannot determine variable parts of the code flow.

On the other hand, runtime analysis is always limited to the code executed, since only this is what is inspected. Consider the following code example:

```
void vulnerable_func(char *str) {
      char  buf[255];

      if ( check_formatting(str) ) {
            do_something( str );
      } else {
            sprintf(buf,"Error in string: %s\n",str);
            log( buf );
      }
}
```

When doing runtime analysis, normally the correct interaction is used with the target process. In the case of the function above, the vulnerable part is located in the handling of exceptional circumstances, where `check_formatting(str)` is failing.

Runtime analysis tries to regain a level of abstraction from the binary code executed. Manual inspection of all binary code in a target is unfeasible. Therefore, levels of abstraction are required to speed up the process of bug finding. Runtime analysis tries to inspect parts of the code flow and give an output, which abstracts the functionality covered by the binary code.

Library call tracing

Most high-level languages use several levels of abstraction to hide small details of data processing and operating system interfacing from the application developer. The order of abstraction is roughly as follows:

1. OS Kernel Functionality, exposed by Software Interrupts
2. Basic operating system functionality
   a. Exposed by low level libraries (eg. libc)
   b. Exposed by High-level OS API
3. Application framework functionality, exposed by derivable classes

Simple runtime analysis will trap and record all calls to the OS Kernel (1) and report them to the user. This approach is used in many UNIX environments with tools such as "strace" or "truss". On a Windows 32 Platform, this approach is not feasible, since the Kernel API changes with releases and is intentionally not documented.

Library call tracing traps the calls to both low-level libraries as well as high-level OS API libraries and records their arguments. This gives the user a more abstract view on the operation of a specific piece of code. n.bug in particular will set breakpoints on all library calls it is supposed to trace and report the arguments of the calls, both upon entering and exiting the library function.

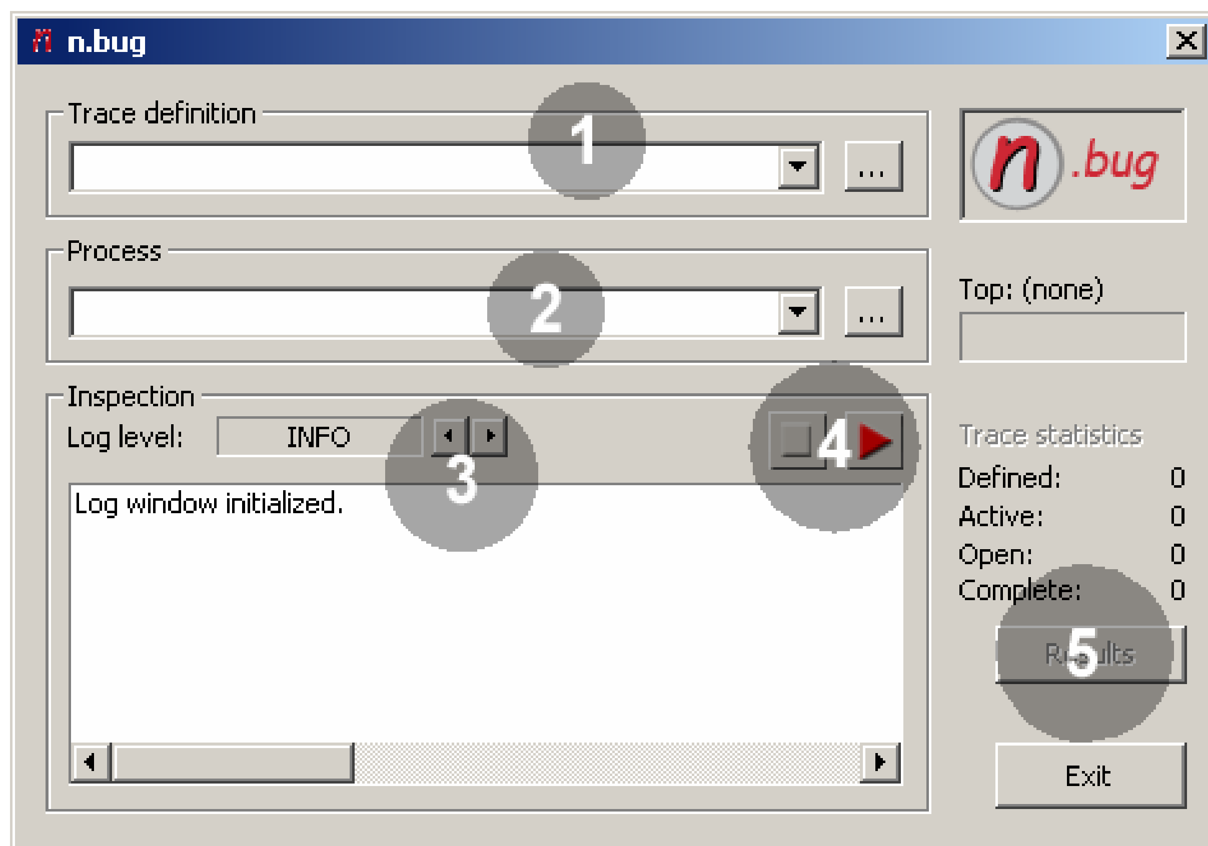
It should be permanently in the mind of the user that library call tracing is not complete. It only covers the code executed and it only covers functionality in the binary implemented by known and traced library functions. Custom code using pointer arithmetic or code using custom libraries with functions not defined to trace are not covered and will not show up in the report.

The user should also keep in mind that only functionality executed **after** the start of a trace is covered. If n.bug is attached to a process (e.g. a server) and no interaction with the server takes place, only the calls in the server's idle loop are traced, which are probably not of any interest. To achieve a maximal effect, the user should try to interact with the target process in

as many different ways as possible. For example, a user authentication should be tried with different usernames and passwords, both correct and wrong, using as many account types as possible.

Using n.bug

When running n.bug, the following user interface appears:



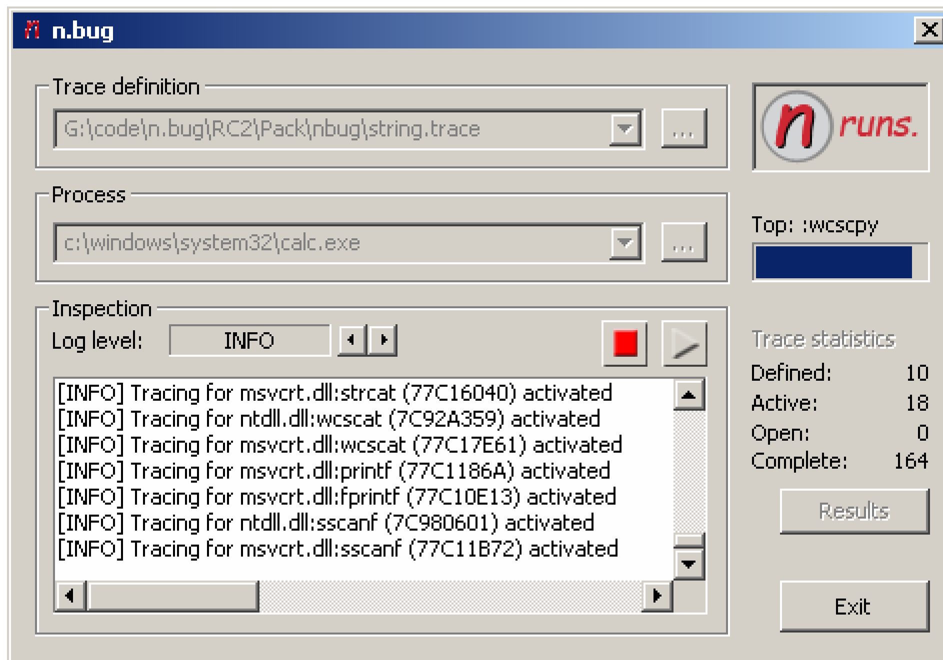
For a trace run, two pieces of information are required. First, a trace definition file (default extension: *.trace) must be selected in trace definition field (Element 1). By pressing the button "…", a standard open dialog is available to select a trace file. The drop down box input field can also be used for direct input of a path to the trace file. If any trace file was selected in previous sessions with n.bug, it will be available for selection in the drop down box.

The process information (Element 2) is used to specify a process to trace. Running processes as well as previously executed command lines are listed in the dropped down part of the drop down box. To attach to a running process or select a previously run command line, open the drop down box and select the appropriate entry. The entries are sorted alphabetically, regardless of their type (running or previously executed). By pressing the "…" button, a new program can be selected for execution. Command line switches necessary for some types of applications can be appended the usual way, separated by spaces.

If desired, the log level of the underlying debugger engine can be adjusted using the spin control (left- and right arrow) marked in the figure as Element 3. A log level of "INFO" should be sufficiently informative. For the analysis process, the information in the log window (Element 3) is not required. Its purpose is to give the advanced user information on what actions are taken in detail during the inspection. If you feel annoyed by the messages, increase the log level by pressing the right pointing arrow one of more times.

The trace is initiated by pressing the red PLAY button (Element 4). Pressing the STOP button (Element 4) will terminate the inspected process in case the user does not terminate it by means of leaving the target program. When the PLAY button is pressed, n.bug will begin

tracing the defined functions and the user interface will disable all controls except the STOP button:

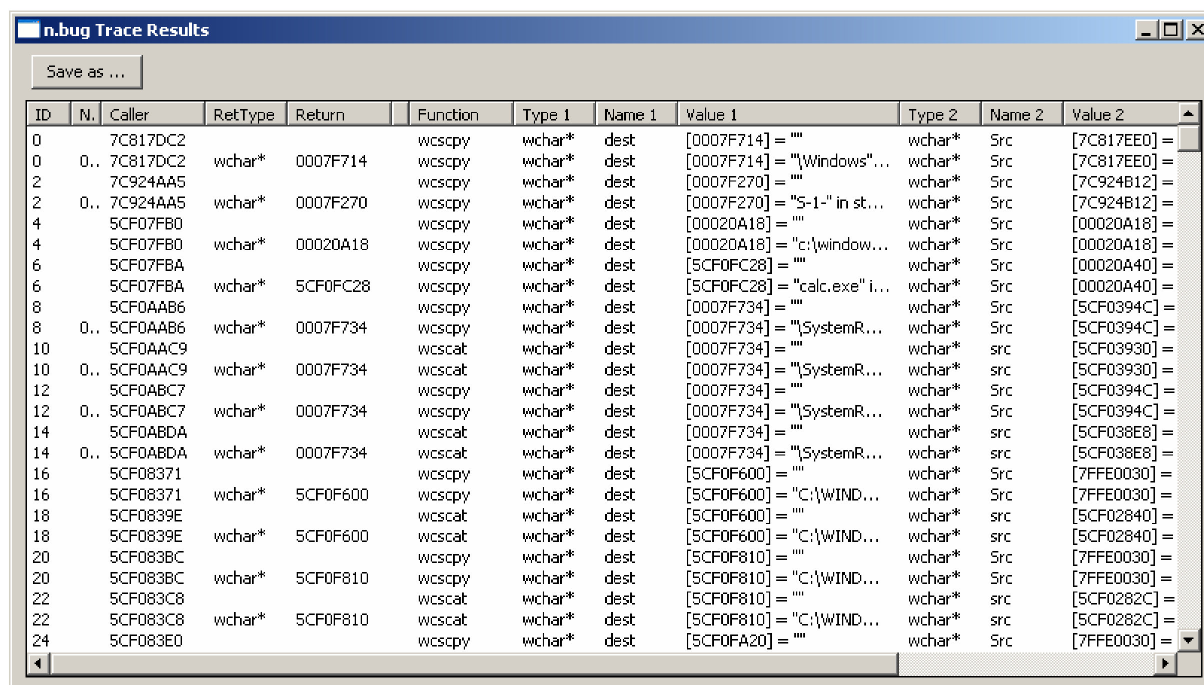


The information on the right hand side gives an overview on the trace details:

- The percent gauge displays the number of calls traced to the most often called function in relation to the total number of completed traces. It also displays the name of the function in question above.
- The "Trace statistics" inform about the current trace:
  - "Defined" contains the number of function definitions found in the trace definition file.
  - "Active" contains the number of functions inspected following the trace definition. This number will often be higher than the "Defined" traces due to the fact that the same functions are exported by more than one library used. See the example above for such a case with the function `sscanf` being exported by ntdll.dll and msvcrt.dll.
  - "Open" contains the number of functions where a call entry was observed but no call exit happened so far. This number will usually be either 0 or 1, except where traced functions call other traced functions.
  - "Complete" represents the number of function calls completely traced so far. This field can be used to control the interaction with the target process. When using or attacking the target process during a trace session, this field should be watched for increases. If this field does not increase, no defined functions where called during the run so far.

Upon completing the trace session, the target process can be either terminated manually by the usual means of exiting a program or by pressing the STOP button. When inspecting services or when not sure, the STOP button should be used, since it terminates the process immediately.

After the trace is completed, the “Results” button (Element 5) is enabled if any function call traces were completed. Pressing the “Results” button will present the result dialog:



ID	N..	Caller	RetType	Return	Function	Type 1	Name 1	Value 1	Type 2	Name 2	Value 2
0		7C817DC2			wcscpy	wchar*	dest	[0007F714] = ""	wchar*	Src	[7C817EE0] =
0	0..	7C817DC2	wchar*	0007F714	wcscpy	wchar*	dest	[0007F714] = "\Windows"...	wchar*	Src	[7C817EE0] =
2		7C924AA5			wcscpy	wchar*	dest	[0007F270] = ""	wchar*	Src	[7C924B12] =
2	0..	7C924AA5	wchar*	0007F270	wcscpy	wchar*	dest	[0007F270] = "S-1-" in st...	wchar*	Src	[7C924B12] =
4		5CF07FB0			wcscpy	wchar*	dest	[00020A18] = ""	wchar*	Src	[00020A18] =
4		5CF07FB0	wchar*	00020A18	wcscpy	wchar*	dest	[00020A18] = "c:\window...	wchar*	Src	[00020A18] =
6		5CF07FBA			wcscpy	wchar*	dest	[5CF0FC28] = ""	wchar*	Src	[00020A40] =
6		5CF07FBA	wchar*	5CF0FC28	wcscpy	wchar*	dest	[5CF0FC28] = "calc.exe" i...	wchar*	Src	[00020A40] =
8		5CF0AAB6			wcscpy	wchar*	dest	[0007F734] = ""	wchar*	Src	[5CF0394C] =
8	0..	5CF0AAB6	wchar*	0007F734	wcscpy	wchar*	dest	[0007F734] = "\SystemR...	wchar*	Src	[5CF0394C] =
10		5CF0AAC9			wcscat	wchar*	dest	[0007F734] = ""	wchar*	src	[5CF03930] =
10	0..	5CF0AAC9	wchar*	0007F734	wcscat	wchar*	dest	[0007F734] = "\SystemR...	wchar*	src	[5CF03930] =
12		5CF0ABC7			wcscpy	wchar*	dest	[0007F734] = ""	wchar*	Src	[5CF0394C] =
12	0..	5CF0ABC7	wchar*	0007F734	wcscpy	wchar*	dest	[0007F734] = "\SystemR...	wchar*	Src	[5CF0394C] =
14		5CF0ABDA			wcscat	wchar*	dest	[0007F734] = ""	wchar*	src	[5CF038E8] =
14	0..	5CF0ABDA	wchar*	0007F734	wcscat	wchar*	dest	[0007F734] = "\SystemR...	wchar*	src	[5CF038E8] =
16		5CF08371			wcscpy	wchar*	dest	[5CF0F600] = ""	wchar*	Src	[7FFE0030] =
16		5CF08371	wchar*	5CF0F600	wcscpy	wchar*	dest	[5CF0F600] = "C:\WIND...	wchar*	Src	[7FFE0030] =
18		5CF0839E			wcscat	wchar*	dest	[5CF0F600] = ""	wchar*	src	[5CF02840] =
18		5CF0839E	wchar*	5CF0F600	wcscat	wchar*	dest	[5CF0F600] = "C:\WIND...	wchar*	src	[5CF02840] =
20		5CF083BC			wcscpy	wchar*	dest	[5CF0F810] = ""	wchar*	Src	[7FFE0030] =
20		5CF083BC	wchar*	5CF0F810	wcscpy	wchar*	dest	[5CF0F810] = "C:\WIND...	wchar*	Src	[7FFE0030] =
22		5CF083C8			wcscat	wchar*	dest	[5CF0F810] = ""	wchar*	src	[5CF0282C] =
22		5CF083C8	wchar*	5CF0F810	wcscat	wchar*	dest	[5CF0F810] = "C:\WIND...	wchar*	src	[5CF0282C] =
24		5CF083E0			wcscpy	wchar*	dest	[5CF0FA20] = ""	wchar*	Src	[7FFE0030] =

Due to the high amount of information recorded for each function call, the columns of the table are not fully extended. Double-clicking on the separating space between two column titles will fully expand the column.

For each traced function call, two rows are added to the table. The column “ID” is increased chronologically to represent the order of the calls. The first row for a function call is the state the arguments had when entering the function. The second row shows the state of the arguments when exiting the function. Each function argument is split into three columns: its type, its name and the value recorded.

The second column contains notes added by the tracing engine. These include information on pointers passed to the function as well as warnings in case of a potential format string vulnerability identified.

Pressing any of the column headers will textually sort the table by this field. This allows sorting the results by the location of pointers, caller addresses, function names and any other information presented.

Pressing the “Save as…” button allows saving the trace results in a human readable text file format for future reference. If the information is not saved, the next trace run or by exiting n.bug will discard the results.

Trace definition files

n.bug uses trace definition files to know the functions to trace and their type and number of arguments. The trace files in general follow a C compatible syntax for function prototypes.

A simple trace definition for the libc call “system” looks like this:

```
int system(char *);
```

A prototype definition must at least contain a return type, a function name, the parentheses and a terminating semicolon.

A more complete specification of the same function to trace would be:

```
int msvcrt:system([in] char *commandline);
```

This includes the exact module where n.bug should look for the function, hereby disabling all other implementations of `system` in other modules. *Do not specify a general trace definition and a module specific together in the same trace file (see* Bugs and glitches*).* It also gives the argument a name, which is then displayed in the result window and the saved report. The `[in]` specification tells n.bug that the parameter is of interest when entering the function only.

The full trace definition syntax is:

```
<return_type> [<module>:]name ( [direction] <arg_type> [name] [, …] );
```

For argument and return types, the following types are valid:

- void
  Function returns void (nothing). This is not valid for an argument type.
- char
  A single character.
- char*
  Pointer to a buffer (potentially) containing a zero terminated string.
- fmtchar*
  Pointer to a buffer containing a zero terminated format string.
- int
  A 32bit integer value.
- int*
  A pointer to a 32bit integer value.
- void*
  An arbitrary 32bit pointer.
- wchar
  A wide character.
- wchar*
  Pointer to a buffer (potentially) containing a zero terminated wide character string.
- fmtwchar*
  Pointer to a buffer containing a zero terminated wide character format string.

Valid direction directives are:

- [in]
  The parameter is valid upon entering the function
- [out]
  The parameter is valid upon exit of the function
- [both]
  The parameter is valid upon entering and exit of the function

n.bug also supports the tracing of arbitrary functions anywhere in the binary's main image or dynamically linked libraries. To specify an arbitrary function, use the address of the first instruction in that function as name. Example:

```
void 0x402342FE(char *overflow);
```

This will trace the function starting at 402342FE and expect one argument (a char pointer) on the stack.

For readability, trace files also support C++ style one-line comments. Everything following a double slash (//) will be considered a comment. Example:

```
// The following function is custom
```

An example of a trace file following the standard libc string handling functions would therefore look like this:

```
// *printf() style functions with buffer overflow potential
int sprintf( [out] char * buf, [in] fmtchar * format );
int swprintf( [out] wchar *buffer, [in] fmtwchar *format);
int vsprintf( [out] char *buffer, [in] fmtchar *format, [in] void *ptr );
// copy functions
char  * strcpy( [out] char *dest, [in] char *Src );
wchar * wcscpy( [out] wchar *dest, [in] wchar *Src );
char  * strcat( [out] char *dest, [in] char *src);
wchar * wcscat( [out] wchar *dest, [in] wchar *src);
```

Note that functions with a variable number of arguments are not supported. You must terminate the function prototype before the first optional argument.

Bugs and glitches

Then n.bug release documented here still suffers from a few bugs itself:

- Function prototypes for OOP methods such as C++ class methods are not supported. To trace these, you must use the arbitrary function notation.
- Definition of multiple functions by the same name in different modules should not be done using the module dependent notation (modulename:function) but by only defining the function itself.
- The trace engine still suffers from a few memory leak issues in the breakpoint handling code, so it is advisable to close and reopen n.bug after about 10 trace runs.