Valgrind overview: Runtime memory checker and a bit more... What can we do with it?

Sergei Trofimovich – slyfox@gentoo.org

MLUG

Mar 30, 2013

The problem

When do we start thinking of weird bug in a program?

The problem

When do we start thinking of weird bug in a program?

- the output is a random garbage

The problem

When do we start thinking of weird bug in a program?

- the output is a random garbage
- random SIGSEGVs

The problem

When do we start thinking of weird bug in a program?

- the output is a random garbage
- random SIGSEGVs
- abort() messages showing "double free()" corruption

Introduction

Meet the valgrind:

Introduction

Meet the valgrind:

- a commandline tool

```
$ valgrind program [args...]
```

Introduction

Meet the valgrind:

- a commandline tool

```
$ valgrind program [args...]
```

- to search bugs in *binary applications*

Introduction

Meet the valgrind:

- a commandline tool

```
$ valgrind program [args...]
```

- to search bugs in *binary applications*
- which contain C/C++ code

Introduction

Meet the valgrind:

- a commandline tool

$ valgrind program [args...]

- to search bugs in *binary applications*
- which contain C/C++ code
- not scripts or bytecode

Introduction

Meet the valgrind:

- a commandline tool

$ valgrind program [args...]

- to search bugs in *binary applications*
- which contain C/C++ code
- not scripts or bytecode

Author: Julian Seward

- compiler writer
- bzip2
- works on current Mozilla's JavaScript engines

First example: no errors

simple.c

```
1 int main() {
2     return 0;
3 }
```

First example: no errors

run-simple.sh

```
make simple CFLAGS=-g
valgrind                                ./simple # to stderr
valgrind --log-file=simple.vglog ./simple # to log
```

First example: no errors

simple.vglog

```
==14290== Memcheck, a memory error detector
==14290== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==14290== Using Valgrind-3.9.0.SVN and LibVEX; rerun with -h for copyright info
==14290== Command: ./simple
==14290== Parent PID: 14287
==14290==
==14290==
==14290== HEAP SUMMARY:
==14290==     in use at exit: 0 bytes in 0 blocks
==14290==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==14290==
==14290== All heap blocks were freed -- no leaks are possible
==14290==
==14290== For counts of detected and suppressed errors, rerun with: -v
==14290== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Simple leak

01-leaky.c

```
1  #include <stdlib.h> /* malloc(), free() */
2  void * leaky() {
3      return malloc (42);
4  }
5  int main() {
6      free ((leaky(), leaky()));
7      return 0;
8  }
9
10 // check as: valgrind --show-reachable=yes \
11 //                    --leak-check=full    \
12 //                    --track-origins=yes  \
13 //                    --quiet
```

Simple leak

01-leaky.vglog

```
==14293== 42 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14293==    at 0x4C2C1DB: malloc (vg_replace_malloc.c:270)
==14293==    by 0x4005C9: leaky (01-leaky.c:3)
==14293==    by 0x4005D9: main (01-leaky.c:6)
==14293==
```

Use of uninitialized memory

02-uninit.c

```
1  int use_uninit (char * uninit) {
2      if (*uninit > 42) /* oh, what will happen ? */
3          return 24;
4      else
5          return 42;
6  }
7  int foo (void) {
8      char garbage;
9      use_uninit (&garbage);
10 }
11 int main() {
12     return foo ();
13 }
```

Use of uninitialized memory

02-uninit.vglog

```
==14297== Conditional jump or move depends on uninitialised value(s)
==14297==    at 0x40052D: use_uninit (02-uninit.c:2)
==14297==    by 0x400550: foo (02-uninit.c:9)
==14297==    by 0x40055B: main (02-uninit.c:12)
==14297==  Uninitialised value was created by a stack allocation
==14297==    at 0x40053D: foo (02-uninit.c:7)
==14297==
```

Out of bounds access

03-oob.c

```c
#include <stdlib.h> /* malloc(), free() */

int main() {
    char * p = (char *) malloc (32);
    *(p + 32 + 129) = '\0'; /* whoops */
    free (p);
    return 0;
}
```

Out of bounds access

03-oob.vglog

```
==14302== Invalid write of size 1
==14302==    at 0x4005DC: main (03-oob.c:5)
==14302==  Address 0x51d80e1 is not stack'd, malloc'd or (recently) free'd
==14302==
```

Helping valgrind

Sometimes compiler optimizations make the final code a complete mess. Here is some tips to make valgrind logs more readable:

Helping valgrind

Sometimes compiler optimizations make the final code a complete mess. Here is some tips to make valgrind logs more readable:

- build your programs/libraries with -g compiler options to emit DWARF debugging symbols (./configure CFLAGS=-g CXXFLAGS=-g)

Helping valgrind

Sometimes compiler optimizations make the final code a complete mess. Here is some tips to make valgrind logs more readable:

- build your programs/libraries with -g compiler options to emit DWARF debugging symbols (./configure CFLAGS=-g CXXFLAGS=-g)
- or install debugging symbols for them (if exist)

Helping valgrind

Sometimes compiler optimizations make the final code a complete mess. Here is some tips to make valgrind logs more readable:

- build your programs/libraries with -g compiler options to emit DWARF debugging symbols (./configure CFLAGS=-g CXXFLAGS=-g)
- or install debugging symbols for them (if exist)
- do not overoptimize things:
  - -O2 optimization reorders and inlines too much code
  - -O1 is usually good-enough
  - -O0 is likely too slow
  - -Og (gcc-4.8 feature) sounds promising

Helping valgrind

Sometimes compiler optimizations make the final code a complete mess. Here is some tips to make valgrind logs more readable:

- build your programs/libraries with -g compiler options to emit DWARF debugging symbols (./configure CFLAGS=-g CXXFLAGS=-g)
- or install debugging symbols for them (if exist)
- do not overoptimize things:
  - -O2 optimization reorders and inlines too much code
  - -O1 is usually good-enough
  - -O0 is likely too slow
  - -Og (gcc-4.8 feature) sounds promising
- -fno-builtin is your friend as valgrind can detect more bugs in mem*() and str*() functions.

Helping valgrind: user's assistance

You can guide valgrind through the source code and tell him the facts about the program he does not know.
There is a mechanism for it: /usr/include/valgrind/*.h

Helping valgrind: an example

04-helpy-uninit.c

```
1  static char tb[32];
2  char * get_temp_buffer () {
3      return tb;
4  }
5  void free_temp_buffer (char * b) { /* superoptimization! */ }
6  int user1 (char * b) {
7      memset (b, 32, 'A');
8      return b[7]; /* a lot of hard work on 'b' */
9  }
10 int user2 (char * b) {
11     /* we forgot this: memset (b, 32, 'B'); */
12     return b[7]; /* a lot of hard work on 'b' */
13 }
14 int main() {
15     char * b; int r1, r2;
16
17     b = get_temp_buffer(); r1 = user1 (b); free_temp_buffer (b);
18     b = get_temp_buffer(); r2 = user2 (b); free_temp_buffer (b);
19     return r1 + r2;
20 }
```

Helping valgrind: an example

04-helpy-uninit.vglog

Helping valgrind: an example

04-helpy-uninit-2.c

```c
#include <valgrind/memcheck.h>
static char tb[32];
char * get_temp_buffer () {
    VALGRIND_MAKE_MEM_UNDEFINED(tb, 32);
    return tb;
}
void free_temp_buffer (char * b) { /* superoptimization! */ }
int user1 (char * b) {
    memset (b, 32, 'A');
    return b[7]; /* a lot of hard work on 'b' */
}
int user2 (char * b) {
    /* we forgot this: memset (b, 32, 'B'); */
    return b[7]; /* a lot of hard work on 'b' */
}
int main() {
    char * b; int r1, r2;

    b = get_temp_buffer(); r1 = user1 (b); free_temp_buffer (b);
    b = get_temp_buffer(); r2 = user2 (b); free_temp_buffer (b);
    return r1 + r2;
}
```

Helping valgrind: an example

04-helpy-uninit-2.vglog

```
==14306== Syscall param exit_group(status) contains uninitialised byte(s)
==14306==    at 0x4EEAA79: _Exit (_exit.c:32)
==14306==    by 0x4E6BC6F: __run_exit_handlers (exit.c:92)
==14306==    by 0x4E6BC94: exit (exit.c:99)
==14306==    by 0x4E5576B: (below main) (libc-start.c:257)
==14306==  Uninitialised value was created by a client request
==14306==    at 0x4007F4: get_temp_buffer (04-helpy-uninit-2.c:4)
==14306==    by 0x400894: main (04-helpy-uninit-2.c:20)
==14306==
```

More APIs to plug into your code

valgrind-APIs.h

```
1  /* valgrind/memcheck.h */
2  VALGRIND_MAKE_MEM_NOACCESS(_qzz_addr,_qzz_len)
3  VALGRIND_MAKE_MEM_UNDEFINED(_qzz_addr,_qzz_len)
4  VALGRIND_MAKE_MEM_DEFINED(_qzz_addr,_qzz_len)
5  VALGRIND_CREATE_BLOCK(_qzz_addr,_qzz_len, _qzz_desc)
6  VALGRIND_CHECK_MEM_IS_ADDRESSABLE(_qzz_addr,_qzz_len)
7  VALGRIND_CHECK_MEM_IS_DEFINED(_qzz_addr,_qzz_len)
8  VALGRIND_CHECK_VALUE_IS_DEFINED(__lvalue)
9  VALGRIND_COUNT_LEAKS(leaked, dubious, reachable, suppressed)
10 /* valgrind/valgrind.h */
11 RUNNING_ON_VALGRIND
12 VALGRIND_DISCARD_TRANSLATIONS(_qzz_addr,_qzz_len)
13 VALGRIND_NON_SIMD_CALL0(_qyy_fn)
14 VALGRIND_NON_SIMD_CALL1(_qyy_fn, _qyy_arg1)
15 ...
16 VALGRIND_MALLOCLIKE_BLOCK(addr, sizeB, rzB, is_zeroed)
17 VALGRIND_FREELIKE_BLOCK(addr, rzB)
18 VALGRIND_CREATE_MEMPOOL(pool, rzB, is_zeroed)
19 VALGRIND_DESTROY_MEMPOOL(pool)
20 VALGRIND_MEMPOOL_ALLOC(pool, addr, size)
21 VALGRIND_MEMPOOL_FREE(pool, addr)
22 ...
```

Bugs in the wild

Recently found bugs:

- cvsps: invalid handling of external modules
- btrfs-progs: invalid checksums for built data
- cvs: massive memory leak on long sessions

Magic!

What a great tool!

Or not?

Well... there is something I have hidden from you.

Generic instrumentation framework

How does *valgrind* work internally?

- userspace JIT compiler with full CPU emulation (just like qemu in binary translation mode)
- uses VEX library to decoding guest code CPU instructions and assembling host code
- uses coregrind library to emulate operating system syscalls

Portability

As valgrind goes down to syscall instructions it needs to know the syscall ABI, signal ABI, etc. of host and guest (emulated) OSes.
Thus valgrind:

Portability

As valgrind goes down to syscall instructions it needs to know the syscall ABI, signal ABI, etc. of host and guest (emulated) OSes.
Thus valgrind:

- won't work on windows in it's current form as there is no documented syscall ABI.
  But!

Portability

As valgrind goes down to syscall instructions it needs to know the syscall ABI, signal ABI, etc. of host and guest (emulated) OSes.
Thus valgrind:

- won't work on windows in it's current form as there is no documented syscall ABI.
  But! There is some gross hacks to run wine under valgrind to unstrument PE files. Scary :]

Portability

As valgrind goes down to syscall instructions it needs to know the syscall ABI, signal ABI, etc. of host and guest (emulated) OSes. Thus valgrind:

- won't work on windows in it's current form as there is no documented syscall ABI.
  But! There is some gross hacks to run wine under valgrind to unstrument PE files. Scary :]
- works poorly on rare OSen (like various BSDs), but it's a matter of adding some syscall effect definition (some lines of code to valgrind/coregrind/m_syswrap/*). Worth looking at valgrind-freebsd.
- ported to relatively popular CPU architectures:
  - IA-32 (i386)
  - AMD64
  - ARM
  - PPC (PPC64)
  - S390X
  - MIPS

Other tools

By default valgrind runs the memcheck tool (valgrind –tool=memcheck).
VEX + coregrind is quite generic framework for runtimes exploration. Other tools shipped with valgrind are:

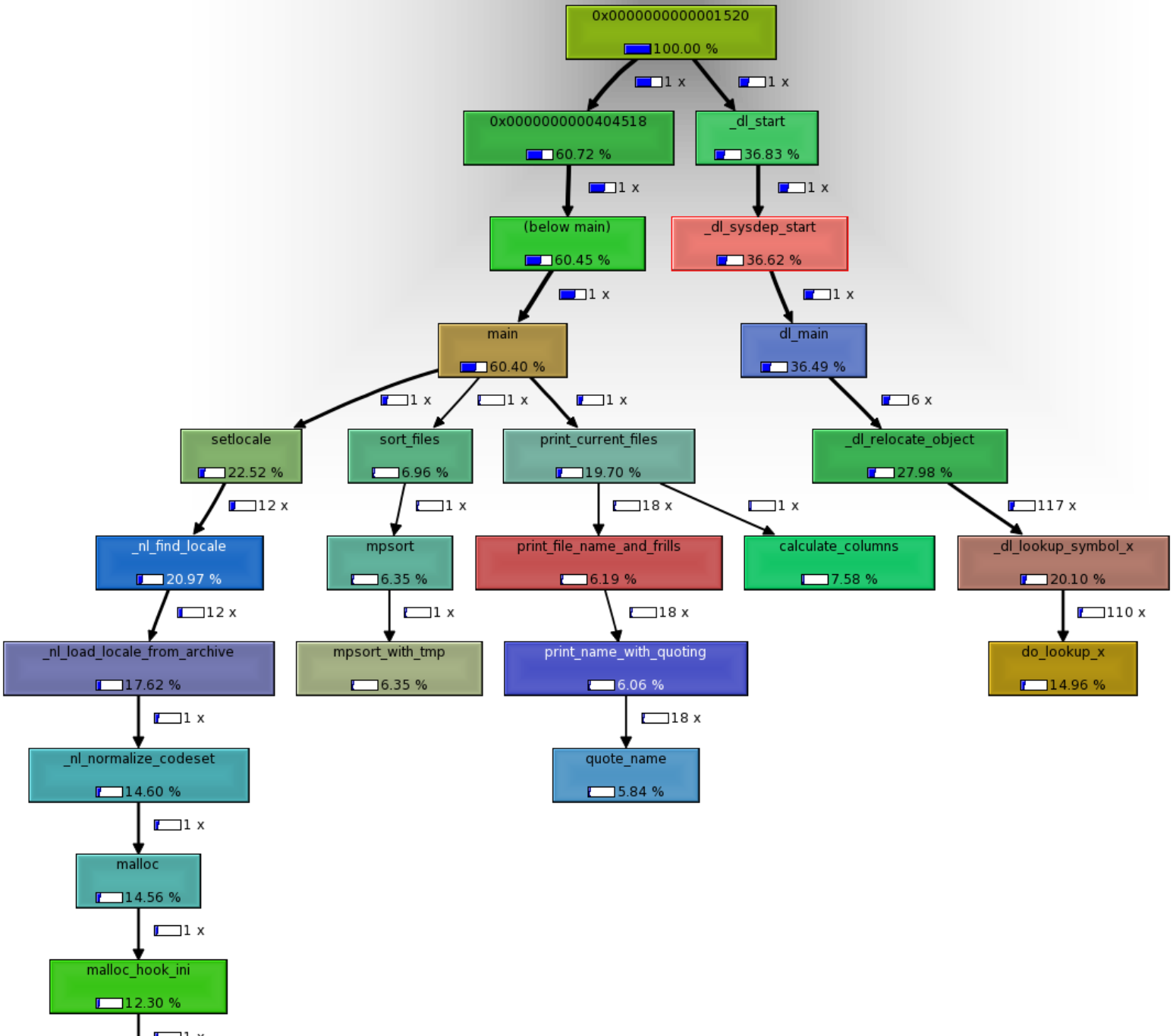
- none - does nothing (useful to measure emulation overlead)
- memcheck - default tool to check for common memory errors
- helgrind - data race detection tool for multithreaded apps using POSIX threads for synchronization
- drd - another data race detector
- cachegrind - l2 cache hit/miss profiler
- callgrind - function call and instruction cost profiler

Other tools: callgrind

An example callgraph of

```
valgrind —-tool=callgrind ls
```

is...


0x0000000000001520
100.00 %
1 x
1 x
0x0000000000404518
60.72 %
_dl_start
36.83 %
1 x
1 x
(below main)
60.45 %
_dl_sysdep_start
36.62 %
1 x
1 x
main
60.40 %
dl_main
36.49 %
1 x
1 x
1 x
6 x
setlocale
22.52 %
sort_files
6.96 %
print_current_files
19.70 %
_dl_relocate_object
27.98 %
12 x
1 x
18 x
1 x
117 x
_nl_find_locale
20.97 %
mpsort
6.35 %
print_file_name_and_frills
6.19 %
calculate_columns
7.58 %
_dl_lookup_symbol_x
20.10 %
12 x
1 x
18 x
110 x
_nl_load_locale_from_archive
17.62 %
mpsort_with_tmp
6.35 %
print_name_with_quoting
6.06 %
do_lookup_x
14.96 %
1 x
18 x
_nl_normalize_codeset
14.60 %
quote_name
5.84 %
1 x
malloc
14.56 %
1 x
malloc_hook_ini
12.30 %

Thank you!

Any questions?