代码测试、调试与优化

2018-02-24 15:41 更新

代码测试、调试与优化

前言

代码写完以后往往要做测试（或验证）、调试，可能还要优化。

- 关于测试（或验证）

通常对应着两个英文单词 `Verification` 和 `Validation` ，在资料 [1] 中有关于这个的定义和一些深入的讨论，在资料 [2] 中，很多人给出了自己的看法。但是正如资料 [2] 提到的：

> The differences between verification and validation are unimportant except to the theorist; practitioners use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required.

所以，无论测试（或验证）目的都是为了让软件的功能能够达到需求。测试和验证通常会通过一些形式化（貌似可以简单地认为有数学根据的）或者非形式化的方法去验证程序的功能是否达到要求。

- 关于调试

而调试对应英文 debug，debug 叫“驱除害虫”，也许一个软件的功能达到了要求，但是可能会在测试或者是正常运行时出现异常，因此需要处理它们。

- 关于优化

debug 是为了保证程序的正确性，之后就需要考虑程序的执行效率，对于存储资源受限的嵌入式系统，程序的大小也可能是优化的对象。

很多理论性的东西实在没有研究过，暂且不说吧。这里只是想把一些需要动手实践的东西先且记录和总结一下，另外很多工具在这里都有提到和罗列，包括 Linux 内核调试相关的方法和工具。关于更详细更深入的内容还是建议直接看后面的参考资料为妙。

下面的所有演示在如下环境下进行：

```
$ uname -a
Linux falcon 2.6.22-14-generic #1 SMP Tue Feb 12 07:42:25 UTC 2008 i686 GNU/Linux
$ echo $SHELL
/bin/bash
$ /bin/bash --version | grep bash
GNU bash, version 3.2.25(1)-release (i486-pc-linux-gnu)
$ gcc --version | grep gcc
gcc (GCC) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
$ cat /proc/cpuinfo | grep "model name"
model name      : Intel(R) Pentium(R) 4 CPU 2.80GHz
```

代码测试

代码测试有很多方面，例如运行时间、函数调用关系图、代码覆盖度、性能分析（Profiling）、内存访问越界（Segmentation Fault）、缓冲区溢出（Stack Smashing 合法地进行非法的内存访问？所以很危险）、内存泄露（Memory Leak）等。

测试程序的运行时间 time

Shell 提供了内置命令 `time` 用于测试程序的执行时间，默认显示结果包括三部分：实际花费时间（real time）、用户空间花费时间（user time）和内核空间花费时间（kernel time）。

```
$ time pstree 2>&1 >/dev/null

real    0m0.024s
```

```
user    0m0.008s
sys     0m0.004s
```

`time` 命令给出了程序本身的运行时间。这个测试原理非常简单，就是在程序运行（通过 `system` 函数执行）前后记录了系统时间（用 `times` 函数），然后进行求差就可以。如果程序运行时间很短，运行一次看不到效果，可以考虑采用测试纸片厚度的方法进行测试，类似把很多纸张叠到一起来测试纸张厚度一样，我们可以让程序运行很多次。

如果程序运行时间太长，执行效率很低，那么得考虑程序内部各个部分的执行情况，从而对代码进行可能的优化。具体可能会考虑到这两点：

对于 C 语言程序而言，一个比较宏观的层次性的轮廓（profile）是函数调用图、函数内部的条件分支构成的语句块，然后就是具体的语句。把握好这样一个轮廓后，就可以有针对性地去关注程序的各个部分，包括哪些函数、哪些分支、哪些语句最值得关注（执行次数越多越值得优化，术语叫 hotspots）。

对于 Linux 下的程序而言，程序运行时涉及到的代码会涵盖两个空间，即用户空间和内核空间。由于这两个空间涉及到地址空间的隔离，在测试或调试时，可能涉及到两个空间的工具。前者绝大多数是基于 `Gcc` 的特定参数和系统的 `ptrace` 调用，而后者往往实现为内核的补丁，它们在原理上可能类似，但实际操作时后者显然会更麻烦，不过如果你不去 hack 内核，那么往往无须关心后者。

函数调用关系图 calltree

`calltree` 可以非常简单方便地反应一个项目的函数调用关系图，虽然诸如 `gprof` 这样的工具也能做到，不过如果仅仅要得到函数调用图， `calltree` 应该是更好的选择。如果要产生图形化的输出可以使用它的 `-dot` 参数。从这里可以下载到它。

这里是一份基本用法演示结果：

```
$ calltree -b -np -m *.c
main:
|   close
|   commitchanges
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   ftruncate
|   |   lseek
|   |   write
|   ferr
|   getmemorysize
|   modifyheaders
|   open
|   printf
|   readelfheader
```

```
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   read
|   readphdrtable
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   malloc
|   |   read
|   truncatezeros
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   lseek
|   |   read$
```

这样一份结果对于“反向工程”应该会很有帮助，它能够呈现一个程序的大体结构，对于阅读和分析源代码来说是一个非常好的选择。虽然 `cscope` 和 `ctags` 也能够提供一个函数调用的“即时”（在编辑 Vim 的过程中进行调用）视图（view），但是 `calltree` 却给了我们一个宏观的视图。

不过这样一个视图只涉及到用户空间的函数，如果想进一步给出内核空间的宏观视图，那么 `strace`，`KFT` 或者 `Ftrace` 就可以发挥它们的作用。另外，该视图也没有给出库中的函数，如果要跟踪呢？需要 `ltrace` 工具。

另外发现 `calltree` 仅仅给出了一个程序的函数调用视图，而没有告诉我们各个函数的执行次数等情况。如果要关注这些呢？我们有 `gprof`。

性能测试工具 gprof & kprof

参考资料[3]详细介绍了这个工具的用法，这里仅挑选其中一个例子来演示。`gprof` 是一个命令行的工具，而 KDE 桌面环境下的 `kprof` 则给出了图形化的输出，这里仅演示前者。

首先来看一段代码（来自资料[3]），算 `Fibonacci` 数列的，

```
#include <stdio.h>

int fibonacci(int n);

int main (int argc, char **argv)
{
    int fib;
    int n;
```

```
    for (n = 0; n <= 42; n++) {
        fib = fibonacci(n);
        printf("fibonnaci(%d) = %d\n", n, fib);
    }

    return 0;
}

int fibonacci(int n)
{
    int fib;

    if (n <= 0) {
        fib = 0;
    } else if (n == 1) {
        fib = 1;
    } else {
        fib = fibonacci(n -1) + fibonacci(n - 2);
    }

    return fib;
}
```

通过 `calltree` 看看这段代码的视图，

```
$ calltree -b -np -m *.c
main:
|   fibonacci
|   |   fibonacci ....
|   printf
```

可以看出程序主要涉及到一个 `fibonacci` 函数，这个函数递归调用自己。为了能够使用 `gprof` ，需要编译时加上 `-pg` 选项，让 `Gcc` 加入相应的调试信息以便 `gprof` 能够产生函数执行情况的报告。

```
$ gcc -pg -o fib fib.c
$ ls
fib  fib.c
```

运行程序并查看执行时间，

```
$ time ./fib
fibonnaci(0) = 0
fibonnaci(1) = 1
fibonnaci(2) = 1
fibonnaci(3) = 2
...
fibonnaci(41) = 165580141
fibonnaci(42) = 267914296

real    1m25.746s
user    1m9.952s
sys     0m0.072s
$ ls
fib  fib.c  gmon.out
```

上面仅仅选取了部分执行结果，程序运行了 1 分多钟，代码运行以后产生了一个 `gmon.out` 文件，这个文件可以用于 `gprof` 产生一个相关的性能报告。

```
$ gprof  -b ./fib gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 96.04     14.31    14.31       43   332.80   332.80  fibonacci
  4.59     14.99     0.68                             main

                     Call graph


granularity: each sample hit covers 2 byte(s) for 0.07% of 14.99 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.68   14.31                 main [1]
               14.31    0.00      43/43          fibonacci [2]
-----------------------------------------------
                            2269806252             fibonacci [2]
               14.31    0.00      43/43          main [1]
[2]     95.4   14.31    0.00      43+2269806252 fibonacci [2]
                            2269806252             fibonacci [2]
-----------------------------------------------
```

```
Index by function name

   [2] fibonacci               [1] main
```

从这份结果中可观察到程序中每个函数的执行次数等情况，从而找出值得修改的函数。在对某些部分修改之后，可以再次比较程序运行时间，查看优化结果。另外，这份结果还包含一个特别有用的东西，那就是程序的动态函数调用情况，即程序运行过程中实际执行过的函数，这和 `calltree` 产生的静态调用树有所不同，它能够反应程序在该次执行过程中的函数调用情况。而如果想反应程序运行的某一时刻调用过的函数，可以考虑采用 `gdb` 的 `backtrace` 命令。

类似测试纸片厚度的方法， `gprof` 也提供了一个统计选项，用于对程序的多次运行结果进行统计。另外， `gprof` 有一个 KDE 下图形化接口 `kprof` ，这两部分请参考资料[3]。

对于非 KDE 环境，可以使用 Gprof2Dot 把 `gprof` 输出转换成图形化结果。

关于 `dot` 格式的输出，也可以可以考虑通过 `dot` 命令把结果转成 `jpg` 等格式，例如：

```
$ dot -Tjpg test.dot -o test.jp
```

`gprof` 虽然给出了函数级别的执行情况，但是如果想关心具体哪些条件分支被执行到，哪些语句没有被执行，该怎么办?

代码覆盖率测试 gcov & ggcov

如果要使用 `gcov` ，在编译时需要加上这两个选项 `-fprofile-arcs -ftest-coverage` ，这里直接用之前的 `fib.c` 做演示。

```
$ ls
fib.c
$ gcc -fprofile-arcs -ftest-coverage -o fib fib.c
$ ls
fib  fib.c  fib.gcno
```

运行程序，并通过 `gcov` 分析代码的覆盖度：

```
$ ./fib
$ gcov fib.c
File 'fib.c'
Lines executed:100.00% of 12
fib.c:creating 'fib.c.gcov'
```

12 行代码 100% 被执行到，再查看分支情况，

```
$ gcov -b fib.c
File 'fib.c'
Lines executed:100.00% of 12
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 4
fib.c:creating 'fib.c.gcov'
```

发现所有函数，条件分支和语句都被执行到，说明代码的覆盖率很高，不过资料[3] `gprof` 的演示显示代码的覆盖率高并不一定说明代码的性能就好，因为那些被覆盖到的代码可能能够被优化成性能更高的代码。那到底哪些代码值得被优化呢？执行次数最多的，另外，有些分支虽然都覆盖到了，但是这个分支的位置可能并不是理想的，如果一个分支的内容被执行的次数很多，那么把它作为最后一个分支的话就会浪费很多不必要的比较时间。因此，通过覆盖率测试，可以尝试着剔除那些从未执行过的代码或者把那些执行次数较多的分支移动到较早的条件分支里头。通过性能测试，可以找出那些值得优化的函数、分支或者是语句。

如果使用 `-fprofile-arcs -ftest-coverage` 参数编译完代码，可以接着用 `-fbranch-probabilities` 参数对代码进行编译，这样，编译器就可以对根据代码的分支测试情况进行优化。

```
$ wc -c fib
16333 fib
$ ls fib.gcda  #确保fib.gcda已经生成，这个是运行fib后的结果
fib.gcda
$ gcc -fbranch-probabilities -o fib fib.c #再次运行
$ wc -c fib
6604 fib
$ time ./fib
...
real    0m21.686s
user    0m18.477s
sys     0m0.008s
```

可见代码量减少了，而且执行效率会有所提高，当然，这个代码效率的提高可能还跟其他因素有关，比如 `Gcc` 还优化了一些跟平台相关的指令。

如果想看看代码中各行被执行的情况，可以直接看 `fib.c.gcov` 文件。这个文件的各列依次表示执行次数、行号和该行的源代码。次数有三种情况，如果一直没有执行，那么用 `####` 表示；如果该行是注释、函数声明等，用 `-` 表示；如果是纯粹的代码行，那么用执行次数表示。这样我们就可以直接分析每一行的执行情况。

`gcov` 也有一个图形化接口 `ggcov` ，是基于 `gtk+` 的，适合 Gnome 桌面的用户。

现在都已经关注到代码行了，实际上优化代码的前提是保证代码的正确性，如果代码还有很多 bug，那么先要 debug。不过下面的这些 "bug" 用普通的工具确实不太方便，虽然可能，不过这里还是把它们归结为测试的

内容，并且这里刚好承接上 `gcov` 部分， `gcov` 能够测试到每一行的代码覆盖情况，而无论是内存访问越界、缓冲区溢出还是内存泄露，实际上是发生在具体的代码行上的。

内存访问越界 catchsegv, libSegFault.so

"Segmentation fault" 是很头痛的一个问题，估计“纠缠”过很多人。这里仅仅演示通过 `catchsegv` 脚本测试段错误的方法，其他方法见后面相关资料。

`catchsegv` 利用系统动态链接的 `PRELOAD` 机制（请参考 `man ld-linux` ），把库 `/lib/libSegFault.so` 提前 load 到内存中，然后通过它检查程序运行过程中的段错误。

```
$ cat test.c
#include <stdio.h>

int main(void)
{
    char str[10];

        sprintf(str, "%s", 111);

        printf("str = %s\n", str);
        return 0;
}
$ make test
$ LD_PRELOAD=/lib/libSegFault.so ./test  #等同于catchsegv ./test
*** Segmentation fault
Register dump:

 EAX: 0000006f   EBX: b7eecff4   ECX: 00000003   EDX: 0000006f
 ESI: 0000006f   EDI: 0804851c   EBP: bff9a8a4   ESP: bff9a27c

 EIP: b7e1755b   EFLAGS: 00010206

 CS: 0073   DS: 007b   ES: 007b   FS: 0000   GS: 0033   SS: 007b

 Trap: 0000000e   Error: 00000004   OldMask: 00000000
 ESP/signal: bff9a27c   CR2: 0000006f

Backtrace:
/lib/libSegFault.so[0xb7f0604f]
[0xffffe420]
/lib/tls/i686/cmov/libc.so.6(vsprintf+0x8c)[0xb7e0233c]
/lib/tls/i686/cmov/libc.so.6(sprintf+0x2e)[0xb7ded9be]
```

```
./test[0x804842b]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe0)[0xb7dbd050]
./test[0x8048391]
...
```

从结果中可以看出，代码的 `sprintf` 有问题。经过检查发现它把整数当字符串输出，对于字符串的输出，需要字符串的地址作为参数，而这里的 `111` 则刚好被解释成了字符串的地址，因此 `sprintf` 试图访问 `111` 这个地址，从而发生了非法访问内存的情况，出现 “Segmentation Fault”。

缓冲区溢出 libsafe.so

缓冲区溢出是堆栈溢出（Stack Smashing），通常发生在对函数内的局部变量进行赋值操作时，超出了该变量的字节长度而引起对栈内原有数据（比如 eip，ebp 等）的覆盖，从而引发内存访问越界，甚至执行非法代码，导致系统崩溃。关于缓冲区的详细原理和实例分析见《缓冲区溢出与注入分析》。这里仅仅演示该资料中提到的一种用于检查缓冲区溢出的方法，它同样采用动态链接的 `PRELOAD` 机制提前装载一个名叫 `libsafe.so` 的库，可以从这里获取它，下载后，再解压，编译，得到 `libsafe.so` ，

下面，演示一个非常简单的，但可能存在缓冲区溢出的代码，并演示 `libsafe.so` 的用法。

```
$ cat test.c
$ make test
$ LD_PRELOAD=/path/to/libsafe.so ./test ABCDEFGHIJKLMN
ABCDEFGHIJKLMN
*** stack smashing detected ***: ./test terminated
Aborted (core dumped)
```

资料[7]分析到，如果不能够对缓冲区溢出进行有效的处理，可能会存在很多潜在的危险。虽然 `libsafe.so` 采用函数替换的方法能够进行对这类 Stack Smashing 进行一定的保护，但是无法根本解决问题，alert7 大虾在资料[10]中提出了突破它的办法，资料[1111]提出了另外一种保护机制。

内存泄露 Memwatch, Valgrind, mtrace

堆栈通常会被弄在一起叫，不过这两个名词却是指进程的内存映像中的两个不同的部分，栈（Stack）用于函数的参数传递、局部变量的存储等，是系统自动分配和回收的；而堆（heap）则是用户通过 `malloc` 等方式申请而且需要用户自己通过 `free` 释放的，如果申请的内存没有释放，那么将导致内存泄露，进而可能导致堆的空间被用尽；而如果已经释放的内存再次被释放（double-free）则也会出现非法操作。如果要真正理解堆和栈的区别，需要理解进程的内存映像，请参考《缓冲区溢出与注入分析》

这里演示通过 `Memwatch` 来检测程序中可能存在内存泄露，可以从这里下载到这个工具。使用这个工具的方式很简单，只要把它链接（ld）到可执行文件中去，并在编译时加上两个宏开关 `-DMEMWATCH -DMW_STDIO` 。这里演示一个简单的例子。

```
$ cat test.c
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"

int main(void)
{
    char *ptr1;
    char *ptr2;

    ptr1 = malloc(512);
    ptr2 = malloc(512);

    ptr2 = ptr1;
    free(ptr2);
    free(ptr1);
}
$ gcc -DMEMWATCH -DMW_STDIO test.c memwatch.c -o test
$ cat memwatch.log
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Sat Mar  1 07:34:33 2008

Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32

double-free: <4> test.c(15), 0x80517e4 was freed from test.c(14)

Stopped at Sat Mar  1 07:34:33 2008

unfreed: <2> test.c(11), 512 bytes at 0x8051a14         {FE FE FE FE FE FE FE FE FE
FE FE FE FE FE FE FE ................}

Memory usage statistics (global):
 N)umber of allocations made: 2
 L)argest memory usage      : 1024
 T)otal of all alloc() calls: 1024
 U)nfreed bytes totals      : 512
```

通过测试，可以看到有一个 512 字节的空间没有被释放，而另外 512 字节空间却被连续释放两次（double-free）。 Valgrind 和 mtrace 也可以做类似的工作，请参考资料[4]，[5]和 mtrace 的手册。

代码调试

调试的方法很多，调试往往要跟踪代码的运行状态， `printf` 是最基本的办法，然后呢？静态调试方法有哪些，非交互的呢？非实时的有哪些？实时的呢？用于调试内核的方法有哪些？有哪些可以用来调试汇编代码呢？

静态调试：printf + gcc -D（打印程序中的变量）

利用 `Gcc` 的宏定义开关（ `-D` ）和 `printf` 函数可以跟踪程序中某个位置的状态，这个状态包括当前一些变量和寄存器的值。调试时需要用 `-D` 开关进行编译，在正式发布程序时则可把 `-D` 开关去掉。这样做比单纯用 `printf` 方便很多，它可以避免清理调试代码以及由此带来的代码误删除等问题。

```
$ cat test.c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int i = 0;

#ifdef DEBUG
        printf("i = %d\n", i);

        int t;
        __asm__ __volatile__ ("movl %%ebp, %0;":"=r"(t)::"%ebp");
        printf("ebp = 0x%x\n", t);
#endif

        _exit(0);
}
$ gcc -DDEBUG -g -o test test.c
$ ./test
i = 0
ebp = 0xbfb56d98
```

上面演示了如何跟踪普通变量和寄存器变量的办法。跟踪寄存器变量采用了内联汇编。

不过，这种方式不够灵活，我们无法“即时”获取程序的执行状态，而 `gdb` 等交互式调试工具不仅解决了这样的问题，而且通过把调试器拆分成调试服务器和调试客户端适应了嵌入式系统的调试，另外，通过预先设置断点以及断点处需要收集的程序状态信息解决了交互式调试不适应实时调试的问题。

交互式的调试（动态调试）：gdb（支持本地和远程）/ald（汇编指令级别的调试）

嵌入式系统调试方法 gdbserver/gdb

估计大家已经非常熟悉 GDB（Gnu DeBugger）了，所以这里并不介绍常规的 `gdb` 用法，而是介绍它的服务器 / 客户（ `gdbserver/gdb` ）调试方式。这种方式非常适合嵌入式系统的调试，为什么呢？先来看看这个：

```
$ wc -c /usr/bin/gdbserver
56000 /usr/bin/gdbserver
$ which gdb
/usr/bin/gdb
$ wc -c /usr/bin/gdb
2557324 /usr/bin/gdb
$ echo "(2557324-56000)/2557324"  | bc -l
.97810210986171482377
```

`gdb` 比 `gdbserver` 大了将近 97%，如果把整个 `gdb` 搬到存储空间受限的嵌入式系统中是很不合适的，不过仅仅 5K 左右的 `gdbserver` 即使在只有 8M Flash 卡的嵌入式系统中也都足够了。所以在嵌入式开发中，我们通常先在本地主机上交叉编译好 `gdbserver/gdb` 。

如果是初次使用这种方法，可能会遇到麻烦，而麻烦通常发生在交叉编译 `gdb` 和 `gdbserver` 时。在编译 `gdbserver/gdb` 前，需要配置(./configure)两个重要的选项：

- `--host` ，指定 gdb/gdbserver 本身的运行平台，
- `--target` ，指定 gdb/gdbserver 调试的代码所运行的平台，

关于运行平台，通过 `$MACHTYPE` 环境变量就可获得，对于 `gdbserver` ，因为要把它复制到嵌入式目标系统上，并且用它来调试目标平台上的代码，因此需要把 `--host` 和 `--target` 都设置成目标平台；
而 `gdb` 因为还是运行在本地主机上，但是需要用它调试目标系统上的代码，所以需要把 `--target` 设置成目标平台。

编译完以后就是调试，调试时需要把程序交叉编译好，并把二进制文件复制一份到目标系统上，并在本地需要保留一份源代码文件。调试过程大体如下，首先在目标系统上启动调试服务器：

```
$ gdbserver :port /path/to/binary_file
...
```

然后在本地主机上启动gdb客户端链接到 `gdb` 调试服务器， （ `gdbserver_ipaddress` 是目标系统的IP地址，如果目标系统不支持网络，那么可以采用串口的方式，具体看手册）

```
$ gdb -q
(gdb) target remote gdbserver_ipaddress:2345
...
```

其他调试过程和普通的gdb调试过程类似。

汇编代码的调试 ald

用 `gdb` 调试汇编代码貌似会比较麻烦，不过有人正是因为这个原因而开发了一个专门的汇编代码调试器，名字就叫做 `assembly language debugger` ，简称 `ald` ，你可以从这里下载到。

下载后，解压编译，我们来调试一个程序看看。

这里是一段非常简短的汇编代码：

```
.global _start
_start:
        popl %ecx
        popl %ecx
        popl %ecx
        movb $10,12(%ecx)
        xorl %edx, %edx
        movb $13, %dl
        xorl %eax, %eax
        movb $4, %al
        xorl %ebx, %ebx
        int $0x80
        xorl %eax, %eax
        incl %eax
        int $0x80
```

汇编、链接、运行：

```
$ as -o test.o test.s
$ ld -o test test.o
$ ./test "Hello World"
Hello World
```

查看程序的入口地址：

```
$ readelf -h test | grep Entry
  Entry point address:               0x8048054
```

接着用 `ald` 调试：

```
$ ald test
ald> display
```

```
Address 0x8048054 added to step display list
ald> n
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000001 edx = 0x00000000
esp = 0xBFBFDEB4 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x007B es  = 0x007B fs  = 0x0000 gs  = 0x0000
ss  = 0x007B cs  = 0x0073 eip = 0x08048055 eflags = 0x00200292

Flags: AF SF IF ID

Dumping 64 bytes of memory starting at 0x08048054 in hex
08048054:  59 59 59 C6 41 0C 0A 31 D2 B2 0D 31 C0 B0 04 31    YYY.A..1...1...1
08048064:  DB CD 80 31 C0 40 CD 80 00 2E 73 79 6D 74 61 62    ...1.@....symtab
08048074:  00 2E 73 74 72 74 61 62 00 2E 73 68 73 74 72 74    ..strtab..shstrt
08048084:  61 62 00 2E 74 65 78 74 00 00 00 00 00 00 00 00    ab..text........

08048055                      59                    pop ecx
```

可见 `ald` 在启动时就已经运行了被它调试的 `test` 程序，并且进入了程序的入口 `0x8048054` ，紧接着单步执行时，就执行了程序的第一条指令 `popl ecx` 。

`ald` 的命令很少，而且跟 `gdb` 很类似，比如这个几个命令用法和名字都类似 `help,next,continue,set args,break,file,quit,disassemble,enable,disable` 等。名字不太一样但功能对等的有：`examine` 对 `x` ， `enter` 对 `set variable {int} 地址=数据` 。

需要提到的是：Linux 下的调试器包括上面的 `gdb` 和 `ald` ，以及 `strace` 等都用到了 Linux 系统提供的 ptrace() 系统调用，这个调用为用户访问内存映像提供了便利，如果想自己写一个调试器或者想hack一下 `gdb` 和 `ald` ，那么好好阅读资料12和 `man ptrace` 吧。

如果确实需要用gdb调试汇编，可以参考：

- Linux Assembly "Hello World" Tutorial, CS 200
- Debugging your Alpha Assembly Programs using GDB

实时调试：gdb tracepoint

对于程序状态受时间影响的程序，用上述普通的设置断点的交互式调试方法并不合适，因为这种方式将由于交互时产生的通信延迟和用户输入命令的时延而完全改变程序的行为。所以 `gdb` 提出了一种方法以便预先设置断点以及在断点处需要获取的程序状态，从而让调试器自动执行断点处的动作，获取程序的状态，从而避免在断点处出现人机交互产生时延改变程序的行为。

这种方法叫 `tracepoints` （对应 `breakpoint` ），它在 `gdb` 的用户手册里头有详细的说明，见 Tracepoints。

在内核中，有实现了相应的支持，叫 KGTP。

调试内核

虽然这里并不会演示如何去 hack 内核，但是相关的工具还是需要简单提到的，这个资料列出了绝大部分用于内核调试的工具，这些对你 hack 内核应该会有帮助的。

代码优化

这部分暂时没有准备足够的素材，有待进一步完善。

暂且先提到两个比较重要的工具，一个是 Oprofile，另外一个是 Perf。

实际上呢？“代码测试”部分介绍的很多工具是为代码优化服务的，更多具体的细节请参考后续资料，自己做实验吧。

参考资料

- VERIFICATION AND VALIDATION
- difference between verification and Validation
- Coverage Measurement and Profiling(覆盖度测量和性能测试,Gcov and Gprof)
- Valgrind Usage
- Valgrind HOWTO
- Using Valgrind to Find Memory Leaks and Invalid Memory Use
- MEMWATCH
- Mastering Linux debugging techniques
- Software Performance Analysis
- Runtime debugging in embedded systems
- 绕过libsafe的保护--覆盖_dl_lookup_versioned_symbol技术
- 介绍Propolice怎样保护stack-smashing的攻击
- Tools Provided by System：ltrace,mtrace,strace
- Process Tracing Using Ptrace
- Kernel Debugging Related Tools：KGDB, KGOV, KFI/KFT/Ftrace, GDB Tracepoint，UML, kdb
- 用Graphviz 可视化函数调用
- Linux 段错误详解
- 源码分析之函数调用关系绘制系列
- 源码分析：静态分析 C 程序函数调用关系图
- 源码分析：动态分析 C 程序函数调用关系
- 源码分析：动态分析 Linux 内核函数调用关系
- 源码分析：函数调用关系绘制方法与逆向建模

- Linux 下缓冲区溢出攻击的原理及对策
- Linux 汇编语言开发指南
- 缓冲区溢出与注入分析(第一部分：进程的内存映像)
- Optimizing C Code
- Performance programming for scientific computing
- Performance Programming
- Linux Profiling and Optimization
- High-level code optimization
- Code Optimization