

Дао DTrace

Часть II

*Из листа простой бумаги,
Взявши ножницы и клей,
Если хватит вам отваги
Можно сделать пять рублей.*

Из народного фольклора

**Евгений Ильин
Филипп Торчинский**

Появившись всего несколько лет назад в Solaris, ныне DTrace «перебрался» и на другие UNIX-системы. Более года назад был сделан порт для FreeBSD, чуть позже для MacOS X и совсем недавно для QNX. Таким образом, DTrace претендует на то, чтобы стать стандартным средством трассировки для UNIX.

Почему же DTrace оказался настолько привлекательным? Во-первых, подавляющее большинство аналогичных инструментов используют статический код, который, будучи «вживлен» в ключевые места кода системы, позволяет отслеживать ограниченное число событий при минимальном действии на систему. Такой подход (реализованный, к примеру, в KLogger для Linux) позволяет даже довольно точно оценивать накладные расходы, возникающие при использовании данного инструмента, но ограничивает множество системных объектов, доступных для наблюдения. DTrace позволяет существенно расширить «область видимости», используя как статические, так и динамические методы инструментовки кода.

Во-вторых, DTrace позволяет получить информацию, пожалуй, о всех составляющих системы. Скажем, утили-

та truss(1), как и DTrace, тоже позволяет трассировать системные вызовы и сигналы, но ничего более. Для того чтобы посмотреть статистику использования виртуальной памяти или ядра, потребуются утилиты vmstat(1M) или kstat(1M) соответственно. DTrace же собирает все эти утилиты «под одну крышу», попутно решая ещё одну проблему. Если ранее при анализе необходимо было коррелировать вывод статистических утилит чуть ли не «вручную», то теперь это можно запрограммировать в скрипте на D.

Конечно, этими двумя пунктами список не ограничивается. Но раз уж речь зашла о truss(1), стоит упомянуть и о том, что для сбора информации эта утилита пользуется файловой системой proc(4), которая разрабатывалась для традиционных средств отладки. Поэтому зачастую для сбора информации исследуемый процесс останавли-

вается, далее снимается требуемая информация о состоянии, и процесс запускается заново. Если ваш сервер обслуживает биржевые операции, то я бы не советовал использовать такой метод трассировки во время торгов. Что же предлагает DTrace, чтобы разобраться, давайте поговорим...

...еще немного о провайдере syscall

В первой части статьи [1] говорилось про методологию инструментовки (модификация кода системы инструментальными средствами), который используется в провайдере fbt.

Чтобы продемонстрировать тот факт, что методология может быть различной и каждый провайдер DTrace вправе использовать свою технику для инструментовки, посмотрим, что происходит в случае использования провайдера syscall. Для этого нам опять

понадобится помощь уже знакомого модульного отладчика mdb.

Метод, которым пользуется провайдер syscall, чтобы отслеживать входы и выходы из системных вызовов – модификация таблицы системных вызовов. При помощи mdb можно «подсмотреть» за тем, как это происходит. Структура элементов таблицы системных вызовов (struct sysent, ссылка на строчку кода может «поехать» со временем) определена в <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/sys/system.h#305>.

Все примеры в этой статье проверялись в Solaris 10 update 4, так что и в любой свежей версии Solaris Express они тоже работают.

```
# mdb -k

Loading modules: [ unix genunix specs dtrace cpu.AuthenticAMD.15
uppc pcplusmp scsi_vhci zfs random ip hook neti sctp arp usba
sl394 qlc fctl md lofs audiosup spps ipc ptm crypto nfs cpc fcp
fcip loginmux ufs sv nsctl sdbc ii rdc mpt ]

> ::sizeof struct sysent

sizeof (struct sysent) = 0x20

> sysent+7*0x20::array struct sysent 1 |::print struct sysent

{
  sy_narg = '\0'
  sy_flags = 0x1
  sy_call = 0
  sy_lock = 0
  sy_callc = wait
}
```

Включаем датчик на wait (в другом терминале команда #dtrace -n 'syscall::wait:entry') и еще раз смотрим запись в таблице вызовов для wait:

```
> sysent+7*0x20::array struct sysent 1 |::print struct sysent

{
  sy_narg = '\0'
  sy_flags = 0x1
  sy_call = 0
  sy_lock = 0
  sy_callc = dtrace_systrace_syscall
}
```

Замена системного вызова на dtrace_systrace_syscall позволяет передать управление DTrace, который снимет необходимую для трассировки информацию до и после системного вызова, передав управление «настоящему» wait, в соответствующий момент.

Провайдер proc

Этот провайдер предназначен для трассировки системных событий, отображающих процесс исполнения кода (запуск и завершение, отправка и обработка сигналов). Казалось бы, это можно сделать при помощи провайдера syscall и соответствующих датчиков (exec, fork и т. п.). Почему же он понадобился? Дело в том, что управление процессами и сигналами осуществляется не только системными вызовами, а еще и функциями из стандартной библиотеки C (особенно это касается сигналов и легковесных процессов – LWP), поэтому только лишь трассировкой системных вызовов не обойтись. Да и «специализация» провайдера делает его использование более удобным при написании

скриптов. К примеру, датчику create в качестве аргумента передается ссылка на структуру psinfo_t, которая содержит, пожалуй, всю информацию о создаваемом процессе с точки зрения операционной системы.

Провайдер содержит небольшое количество датчиков (список датчиков можно посмотреть командой «dtrace -l -P proc»), из названий очевидно следует их предназначение. В качестве примера посмотрим, каким образом можно отследить цепочку запуска процессов при вызове определенной команды:

```
#pragma D option quiet

proc::exec
{
    self->parent = execname;
}

proc::exec-success
/self->parent != NULL/
{
    @gs[self->parent, execname] = count();
    self->parent = NULL;
}

proc::exec-failure
/self->parent != NULL/
{
    @gf[self->parent, execname] = count();
    self->parent = NULL;
}

END
{
    printa(" %s -> %s (%@d)\n", @gs );
}
```

Пусть скрипт хранится в файле exec-chain.d, в качестве подопытного возьмем команду su:

```
joda@dtrace -s exec-chain.d -c 'su - obi-wan'
obi-wan#exit
```

```
bash -> clear (1)
bash -> cat (1)
bash -> quota (1)
bash -> mail (1)
bash -> sed (1)
clear -> tput (1)
su -> bash (1)
bash -> stty (2)
bash -> expr (20)
```

Как нетрудно заметить, порядок вывода не соответствует хронологическому порядку, поскольку DTrace гарантирует регистрацию события, а порядок вывода на печать – нет, оставляя эту задачу автору. Если важно соблюсти временную последовательность, то можно воспользоваться одной из встроенных переменных для учета времени timestamp, vtimestamp или walltimestamp. Для относительного учета вполне подойдет переменная timestamp, стандартный шаблон её использования: инициализация thread-local-переменной в датчике:

```
BEGIN {
    self->start = timestamp;
}
```

и регистрация времени события относительно начального момента при помощи конструкции:

```
timestamp-self->start
```

после чего отсортировать список событий по времени. Допустим, `exes-chain.d` можно изменить, добавив в кортеж агрегации `@gs` последнюю приведённую конструкцию, добавить вывод отметки времени и передать вывод утилите `sort`.

Этот простой пример демонстрирует очень мощный принцип: вывод скрипта может быть данными, предназначенными для последующей обработки другими утилитами. Более того, вывод скрипта на D может представлять из себя код программы на некотором языке (примеры, где скрипт на D выводит скрипт на Perl, можно легко найти в Интернете). Если в примере `exes-chain.d` (без модификаций с `timestamp`) заменить тело компоненты `END` на:

```
END
{
    printf( "digraph ExecGraph {\n" );
    printa( " \"%s\" -> \"%s\" [weight=%d];\n", @gs );
    printa( " \"%s\" -> \"%s\" ┘
        [color=red,weight=%d];\n",@gf );
    printf( "}\n");
}
```

то после отработки скрипта получим описание графа на языке `dot`, который используется в пакете визуализации графов `Graphviz` (<http://www.graphviz.org/About.php>). Сохранив вывод в файле `graph.dot`, сгенерируем дерево вызовов:

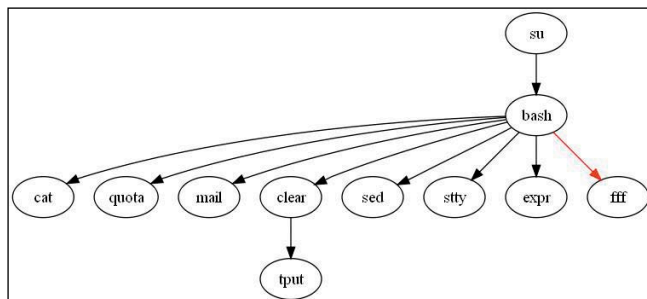
```
#dot -Tjpg graph.dot -o graph.jpg
```

Это даёт нам очень наглядную картинку запускаемых командой `su` процессов (см. **рисунок**). Красными дугами в полученном графе будут обозначаться неудавшиеся запуски. Уверены, что скорей всего команда `su` у вас работает нормально, и красных дуг на картинке не будет. Пришлось специально моделировать ситуацию для того, чтобы получить такой граф.

Трассировку запусков можно ещё получить совсем простой командой в одну строчку (пример взят с http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_One_Liners, где вы найдете массу полезных примеров в одну строчку для различных провайдеров):

```
#dtrace -n 'proc:::exec-success ┘
{ trace(curpsinfo->pr_psargs); }'
```

Этот пример может пригодиться для трассировки удачных запусков в системе, однако мало чего скажет о взаимной зависимости запускаемых приложений. Воспользовавшись же рассмотренным ранее способом визуализации, можно устанавливать не только «родственные» отношения между запущенными процессами, но и даже дерево вызовов функций произвольного приложения.



Дерево вызовов `exes`, сгенерированное утилитой `su`

Почти магия — провайдер `pid`

Это, пожалуй, самый мощный и самый таинственный провайдер в `DTrace`. Он позволяет отслеживать входы и выходы из почти всех функций любого приложения, причём без каких-либо дополнительных манипуляций. Датчики для этого провайдера создаются динамически, поэтому, чтобы посмотреть список, одной лишь опцией `-l` вы не обойдётесь. Необходимо будет указать ещё и приложение, которое будет трассироваться, например, таким образом:

```
#dtrace -P pid'$target' -c dtrace -l
```

Вывод этой команды даст множество датчиков, которые впоследствии могут быть использованы для написания скриптов. Надо сказать, что этот провайдер полезен в первую очередь для разработчиков приложений, но и для системных администраторов он тоже небесполезен, так как даёт возможность увидеть, каким образом происходит взаимодействие приложений со стандартными библиотеками. К примеру, запустив скрипт `libfunc.d`:

```
#!/usr/sbin/dtrace -s
pid$target:$1::entry
{
    @func[probefunc] = count();
}
```

командой: `dtrace -s libfunc.d -c tar libcmd`, мы увидим, сколько раз и какие функции вызываются из динамической библиотеки `libcmd.so.1` при запуске утилиты `tar` без аргументов.

Не менее интересный трюк, который можно проделать при помощи этого провайдера — поиск возможных утечек памяти. Допустим, что в вашем приложении выделение и освобождение памяти происходит только при помощи стандартных вызовов `malloc()` и `free()`. Тогда при помощи примерно такого скрипта:

```
#!/usr/sbin/dtrace -s
pid$target:libc:malloc:entry
{
    ustack();
}
pid$target:libc:malloc:return
{
    printf("alloc: %x\n", arg1);
    @memu[ arg1 ] = count();
}
pid$target:libc:free:entry
{
    printf("free: %x\n", arg0);
    @memu[ arg0 ] = count();
}
END
{
    printa(@par);
}
```

можно отследить, какие выделенные вызовом `malloc()` участки памяти не были впоследствии освобождены при помощи вызова `free()`. Действие `ustack()` на входе в `malloc` позволяет увидеть стек вызовов на момент выделения памяти:

```
malloc: 8069f60
libc.so.1`malloc
libc.so.1`real_gettext_u+0x82
libc.so.1`gettext+0x5a
tar`0x8052f3e
tar`main+0x44
tar`0x80522d6
```

Как можно увидеть в этом выводе, у некоторых функций вместо имени указан некоторый адрес памяти. Настало время немного поговорить о методологии инструментария этого провайдера.

Имена функций, очевидно, берутся из заголовка ELF-файлов (точнее, из символических таблиц разделов .symtab и .dynsym). Следовательно, если после сборки исполняемый файл был почищен утилитой strip, некоторые имена могут навсегда кануть в Лету. Однако, если сборка производилась при помощи последних версий компиляторов Sun Studio, которые создают специальную секцию .SUNW_ldynsym в заголовке ELF-файла, то последствия запуска будут не так катастрофичны для наблюдаемости (если интересны дальнейшие подробности – http://blogs.sun.com/ali/entry/what_is_sunw_ldynsym). Далее, для того, чтобы показать имена функций адреса, на стеке отображаются соответствующие символы из заголовка ELF.

Сказанного уже достаточно, чтобы осознать тот факт, что использование этого провайдера накладно с точки зрения дополнительной нагрузки на систему, особенно если мы имеем дело с большим приложением, в котором количество функций измеряется тысячами. Только лишь динамическое создание датчиков может забрать существенное количество системных ресурсов и привести к дестабилизации системы. Для того чтобы избежать отказа от обслуживания (denial of service), количество датчиков, создаваемых провайдером pid, ограничено.

DTrace для Web 2.0: следим за веб-приложениями на ходу

И девятирусская башня начинается с земли.

Японская пословица

Однажды у нас был случай, когда скрипт на PHP непостижимым образом выдавал лишнюю пустую строку в генерируемой странице. А страница импортировалась в поток RSS у разных хороших людей. То есть должна была, но не импортировалась, потому что лишняя пустая строка все портила. Тогда дело решилось внимательным изучением всех включаемых файлов .php и обнаружением лишней строки. Однако бывают и более сложные случаи, когда при отладке веб-приложений хочется понять, как на самом деле это приложение работает.

Сейчас очень многие веб-сайты, даже простые, строятся на основе системы управления контентом (CMS, content management system). И еще многие другие – на связке PHP-скрипт – база данных. С другой стороны, а именно со стороны клиента, заполнение разнообразных форм на веб-странице обычно связано с кодом на JavaScript, так как это – стандартный способ проверять допустимость значений в полях формы.

Можем ли мы в отладке веб-приложений использовать DTrace для того, чтобы отследить все передаваемые от браузера к базе данных (и обратно) параметры на всем пути их следования через скрипты-обработчики? Если мы используем Solaris – то да. И более того, легко.

Технология DTrace в настоящее время воплощена в Solaris (начиная с Solaris 10) и портирована в Mac OS X Leopard, FreeBSD 6.2 (частично) и QNX Neutrino (QNX6).

В данной статье мы рассматриваем ее применение в Solaris и помним, что приложения третьих компаний (например, Firefox) в виде готового пакета для других систем с поддержкой DTrace могут быть еще недоступны для скачивания. Однако если вы хотите инструментировать любое приложение с открытым кодом, используя DTrace, никто не вправе помешать вам. Описание того, как это сделать с помощью провайдера sdt (statically defined tracing), можно найти на странице http://www.solarisinternals.com/wiki/index.php/DTrace_Topics_USDT.

Приводимые в статье примеры основаны на наборе DTrace Toolkit, который состоит из 386 скриптов (цифра верна для версии DTrace Toolkit 0.99) на языке D и содержит скрипты почти на все случаи жизни, начиная с простых примеров и продолжая трассировкой Java-приложений, о которой можно написать отдельную статью. Скачать DTrace Toolkit можно по адресу <http://www.opensolaris.org/os/community/dtrace/dtracetoolkit>.

Каждый датчик, по срабатыванию которого мы можем выполнять требуемые нам действия в D-скрипте, относится к определенному провайдеру. Системным провайдерам (типа syscall, io и пр.) соответствует одноименный модуль ядра, а за провайдеры, созданные сторонними разработчиками, отвечает модуль ядра sdt. Это вам скорее всего уже известно из предыдущей статьи про DTrace [1] (а может быть, вы это узнали еще раньше?) Теперь мы изучим возможности, которые нам предоставляет провайдер javascript.

Провайдер javascript

В свежих сборках Firefox (по адресу <http://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/contrib/latest-trunk>) в код вставлены датчики DTrace. Из D-скриптов к ним надо обращаться через провайдер javascript. Для экспериментов мы выбрали firefox-3.0a9pre.en-US.solaris11-i386.tar.bz2. Кстати, имейте в виду, что более старые версии Firefox могли включать этот же провайдер под другим именем – mozilla. Что именно позволяет посмотреть провайдер javascript?

```
# dtrace -l -n 'javascript*:::'|more
```

ID	PROVIDER	MODULE	FUNCTION	NAME
72803	javascript2258	libmozjs.so	jsdtrace_execute_done	execute-done
72804	javascript2258	libmozjs.so	js_Execute	execute-done
72805	javascript2258	libmozjs.so	jsdtrace_execute_start	execute-start
72806	javascript2258	libmozjs.so	js_Execute	execute-start
...				

(вывод команды сокращен)

Как видно из вывода dtrace, имя провайдера появляется сопряженным с идентификатором процесса firefox-bin, а датчики вводятся для ряда основных функций. Также есть датчики function-entry, function-info, function-return, function-rval, object-create, object-create-start, object-create-done и object-create-finalize. Набор датчиков в будущем может быть расширен, но в той версии mozilla, которая оказалась в нашем распоряжении, других датчиков не было.

Какой толк можно получить из срабатывания этих датчиков? Во-первых, можно традиционно трассировать выполнение функции, написанной на javascript, и мерить время выполнения разных ее компонент. Во-вторых, уже сейчас доступна экспериментальная функциональность провайдера javascript по трассировке функций с выводом их аргументов.

Детальную информацию об аргументах датчиков, предназначенных для этого, можно почерпнуть на странице <http://news.speeple.com/blogs.sun.com/2007/10/23/dtrace-mozilla-rfe-javascript-tracing-framework-landed.htm>:

```
javascript*::: function-args
```

```
Args: (char *filename, char *classname, char *funcname, int argc,
void *argv, void *argv0, void *argv1, void *argv2, void *argv3,
void *argv4)
```

```
javascript* :::function-rval
```

```
Args: (char *filename, char *classname, char *funcname, int lineno,
void *rval, void *rval0)
```

Датчик function-args служит для показа аргументов вызванной javascript-функции в скрипте, а function-rval для показа возвращаемого функцией значения.

Кстати, если под рукой нет документации, то некоторое представление о том, какие аргументы есть у датчика, и следовательно, какие параметры arg0..argN вам требуется выводить командой printf в скрипте, можно получить, используя ключи -l -v команды dtrace (только давать их надо перед ключом -n, если вы его используете, это важно!)

```
dtrace -l -v -n 'javascript*:::function-args'
```

```
72808 javascript2258 libmozjs.so js_Interpret function-args
```

Probe Description Attributes

```
Identifier Names: Private
Data Semantics: Private
Dependency Class: Unknown
```

Argument Attributes

```
Identifier Names: Private
Data Semantics: Private
Dependency Class: Unknown
```

Argument Types

```
args[0]: char *
args[1]: char *
args[2]: char *
args[3]: int
args[4]: void *
args[5]: void *
args[6]: void *
args[7]: void *
args[8]: void *
args[9]: void *
```

Как видно, датчик function-args может иметь до 9 аргументов, смысл которых пояснен выше, а при определенном воображении о значении аргументов можно догадаться по их типу. Так или иначе, команду dtrace -l -v стоит иметь в виду.

Попробуем выяснить с помощью датчика function-args, какое значение из формы в файле .html передается в написанную нами функцию на javascript. Наш экспериментальный файл и код javascript представляют собой простую форму для ввода единственного поля – даты и проверку на формат даты соответственно:

```
<html>
<head>
<title>Input form</title>

<script language="JavaScript" type="text/javascript">

function testDate(date) {
```

```
    alert(date);
    return(true);
}

function testBox1(form) {
    /* Parsing of date and time */
    IDate = form.date.value;
    if (testDate(IDate)) return (true);
    return (false);
}

function runSubmit (form, button) {
    if (!testBox1(form)) return;
    document.inputform.submit();
    // un-comment to actually submit form
    return;
}

</script>

</head>
<body>

<form name="inputform" action="/cgi-bin/main.pl">
    <input name="date" type="text" size="8"></td>
    <input type="button" name="act" value="Submit"
        onClick="runSubmit(this.form,
this)">
</form>

</body>
</html>
```

Функция testDate в этом примере фактически не проверяет дату, а лишь выдает ее значение в окне сообщения, однако вместо такой заглушки можно написать честную проверку. Для демонстрации работы с dtrace она нам не понадобится. В скрипт /cgi-bin/main.pl передаются введенные данные из поля. Листинг main.pl мы не приводим, так как сейчас он не имеет для нас значения – мы изучаем работу функции проверки данных на стороне клиента, а скрипт main.pl работает на сервере.


Запускаем firefox-bin с включенными в него датчиками, открываем файл с вышеприведенным кодом и переходим к эксперименту:

```
# dtrace -n 'javascript*:::function-args'
/copyinstr(arg2) == "testDate"/ {printf("%s %s
%d %s", copyinstr(arg0), copyinstr(arg2), arg3,
copyinstr(arg5));}
```

```
dtrace: description 'javascript*:::function-args' matched 4 probes
CPU ID FUNCTION:NAME
1 18348 jsdtrace_function_args:function-args file:///export/home/
filip/jstest.html testDate 1 00.00.00
```

Как видно из вывода нашего скрипта, в качестве даты мы вводили значение 00.00.00. Будем надеяться, что настоящая функция проверки даты такого безобразия не пропустит!

Заключение

В этой части статьи мы обсудили визуализацию результатов работы DTrace с помощью программы визуализации графов, но есть и более универсальные средства визуализации, например, Chime и D-Light. За рамками данной статьи остались, кроме этого, ряд применений DTrace, таких, как отладка сценариев на РНР и Java-приложений, а также наблюдение за СУБД и анализ их производительности. Обо всем этом пойдет речь в следующих статьях про DTrace. 

1. Ильин Е. Дао DTrace. //Системный администратор, №12, 2007 г. – С. 6-11.