Learning DTrace

Part 2: Scripts and the D Language[1]

Chip Bennett

편역: vangelis(securityproof@gmail.com)

DTrace에 대한 이 시리즈 중 ***Part 1***[2]에서 필자는 커맨드 라인에서 DTrace를 사용하는 것에 대한 기초들을 다루었다. 커맨드 라인에서 DTrace 명령을 사용하고, 이때 사용하는 아규먼트들이 많아져 명령 라인이 복잡해지면 명령 라인을 더 길게 만드는 것보다는 차라리 스크립트를 사용하는 것이 일반적으로 더 용이할 것이다. 다음은 D 프로그램의 기본적인 레이아웃이다.

```
#!/usr/sbin/dtrace -s

probe-description[, probe-description[, ...]]
/ predicate /
{
   action; [action; [...]]
}
```

이 부분을 하나씩 분석해보자.

1. 첫 라인은 쉘 스크립트[3]에서 볼 수 있는 것과 유사하다. **-s** 옵션은 커맨드 모드가 아니라 스크립트 모드로 실행된다는 것을 DTrace에게 알려주기 위해 사용된 것이다.

2. *probe-description*는 Part 1에서 언급했듯 4개의 tuple probe[4] 로 기술된 것이다. 이 지시자(specifier)들은 **-n** 타입으로 되어 있어야 한다. 즉, D 컴파일러는 단어 하나, 또는 마지막 콜론 다음에 나오는 무엇이든 probe 이름이다(하나의 콜론에 대한 probe-description은 모든

---

[1] (역자 주)이 글을 읽기 전에 Securityproof의 도서관 섹션의 강좌 게시판에 역자가 올린 "Learning DTrace - Part 1: Introduction"를 먼저 읽기를 권합니다.

[2] 각주 1에서 언급한 동일 문서

[3] 이 구조 이면의 원리는 *exec* 시스템 호출이 실행파일을 불러내는(invoke) 방법을 결정하도록 허용하는 것이다. *exec*에게는 모든 것이 어떤 종류의 실행파일인지를 처음에 말해주는 2 바이트의 매직 넘버를 가진 실행파일의 바이너리이다. 일반적인 실행파일의 바이너리들은 그 파일이 ELF 포맷인지, 심볼들이 strip되어 있는지 등을 나타내는 코드를 가지고 있다. 스크립트의 시작부분에 **#!**라는 2 바이트 코드는 이것이 스크립트라는 것을 말해준다. 이것은 Unix에서 모든 스크립트를 사용하는 표준 방법이다.

[4] (역자 주) 각 probe는 4개의 부분(tuple: provider, module, function, name)으로 구성되어 있다

probe와 일치할 것이다). 콤마로 분리된 probe tuple들의 목록을 사용함으로써 하나 이상의 probe-description을 지정할 수 있다.

3. *predicate*는 C 스타일의 관계문(relational expression)이다. 이 라인은 옵션이다. 생략된 predicate는 항상 참(true)인 predicate와 같은 효과를 가지고 있다.

4. 중괄호({})에 둘러싸인 액션들은 만약 (1)삽입된 probe가 시동된다면, 그리고 (2)그 predicate가 참이라면 수행될 액션들이다. 중괄호 안의 action body는 역시 C 스타일로 되어 있지만 어떤 흐름을 통제하는 문장(flow control statement)을 가지고 있지는 않으며, ANSI-C[5]와는 다른 이용 가능한 함수들의 세트를 가지고 있다. 이 액션들 역시 옵션이다. 비어있는 액션 블록은 Part 1에서 기술한 것처럼 단지 기본적인 액션만 수행한다.

probe-descriptions, predicate, 그리고 액션 블록은 probe 구문(*probe clause*)이라는 것을 함께 만든다. 하나의 D 프로그램에는 많은 probe 구문들이 있을 수 있으며, 그들의 probe-description들은 겹칠 수 있다(예를 들어, **syscall:::entry**, **open:entry**, 그리고 **open:**은 **syscall::open:entry** probe의 시동과 모두 일치할 것이다.). probe가 시동할 때 그 probe와 일치하는 구문들만이 실행되고, 그리고 그것들은 D 프로그램에서 등장하는 순서로 실행된다(대응하는 predicate들도 역시 그 구문의 액션 블록이 실행되기 이전에 참이어야 한다는 것을 명심해야 한다.).

예제 스크립트

실제로 스크립트가 어떻게 생겼는지 알아보기 위해 몇 가지 예제 스크립트를 살펴보자. 아주 간단한 스크립트는 여러분의 시스템에 있는 모든 probe들을 삽입(instrument[6])하는 것이다.

```
#!/usr/sbin/dtrace -s

:
{ }
```

---

[5] 최신 버전의 Solaris 10 또는 OpenSolaris에서, 표준 C 문자열 조작 함수들과 유사한 몇 가지 새로운 함수들이 추가되었다. 하지만, 이 새로운 기능들에 대한 문서화는 늦어지고 있다.

[6] (역자 주) Part 1에서는 instrument라는 말을 'insert'와 동일한 의미로 사용된다고 필자가 말하고 있으며, 딱히 우리말로 번역하려면 instrument라는 단어가 가진 '일반적인' 의미를 적용하기가 힘들다. 그래서 역자는 이 글 전체에서 instrument를 '삽입하다'로 계속 사용할 것이다. 'insert(instrument) probe'는 커널이나 사용자 프로그램의 특정 포인트에 probe를 '삽입'하는 것을 의미한다. 번역된 말이 정확하게 그 의미를 살리기에 부족하다면 정확한 문맥을 이해하고 원어를 그대로 사용하는 것이 최선일 것이다.

***DTrace hint**: 이 스크립트를 연구실이나 테스트 시스템에서 독자들이 사용할지 모르겠다. 내부적으로 이 스크립트는 모든 이용 가능한 커널 probe를 활성화한다. DTrace는 아주 효율적이지만 이 스크립트는 약간의 문제를 일으킬 수 있다. 필자는 별다른 문제 없이 여러 번 그것을 사용했지만 조심해야 한다. 이 스크립트를 사용하면서 아마도 접하게 될 다른 하나의 문제는 데이터가 버려지게(drop) 된다는 것이다. Probe가 시동될 때마다 DTrace는 버퍼(consumer와 CPU에 각각 하나씩)에 데이터를 기록한다. 만약 버퍼가 차게 되면 데이터가 버려지게 된다. 여러분들은 연구실이나 테스트 시스템에 **dtrace -n : > /dev/null**를 실행함으로써 이 버려진 것들을 볼 수 있다. 일반적인 trace 출력결과는 /dev/null로 보내질 것이지만, drop 메시지는 표준 에러로 보내진다.*

만약 독자들이 이 스크립트를 실행하려고 한다면 이것을 파일(예를 들어, "intense.d")로 저장한 후 실행 퍼미션(**chmod +x intense.d**)을 준 후 명령 프롬프트에서 실행(**./intense.d**)해야 할 것이다. 다른 스크립트와 마찬가지로 그것을 실행파일[7] 로 변경하는 것보다는 차라리 인터프리터에 아규먼트로 그것을 건넬 수 있을 것이다. 이와 같이 그 스크립트를 불러오기 위해 **dtrace -s intense.d**를 입력한다. 만약 이 방법을 여러분이 사용한다면 preamble(**#!/usr/sbin/dtrace -s**)이 필요하지 않을 것이며, execute bit를 활성화시킬 필요도 없다.

나머지 예에서 필자는 preamble을 보여주지는 않을 것이지만 경우에 따라 필요함을 명심해야 한다.

이제 degenerate 프로그램에 내용을 추가해서 좀더 그럴듯하게 실행되도록 해보자.

```
syscall::exec*:entry
{ }
```

얼마나 많은 probe들을 삽입(**dtrace -ls sample.d**)할지를 지정하기 위해 처음에 **–l** 옵션으로 이 스크립트를 불러내는 것을 잊지 마라. 두 개의 probe(*exec*와 *exece*[8])를 매치시켰어야만 했다. 이것은 probe tuple들을 지정할 때 기본적인 쉘 패턴 매칭을 사용해 이루어진다. 이제 삽입(**dtrace -s sample.d**) 하기 위해 스크립트를 적용해보자. 이 D 프로그램이 나타내는 것을

---

[7] **dtrace** 명령에 대한 아규먼트로서 스크립트를 불러오는 것은 얼마나 많은 probe들을 매칭시켰는지 처음에 확인하기 위해 **-l** 옵션으로 새로운 스크립트를 실행하고자 할 때 더 의미가 있다. **-s** 플래그는 마지막에 위치시켜야 하는데, 이것은 **dtrace**가 **-s** 플래그 다음에 있는 어떤 것이 스크립트의 파일명이라고 간주하기 때문이다. 예를 들어, "intense.d"라는 스크립트가 삽입했을 probe들의 목록을 열거하기 위해서는 **dtrace -ls intense.d**와 같이 실행하면 된다.

[8] **exec** 시스템 호출은 하나의 프로세스가 또 다른 하나의 프로세스를 호출하는 메커니즘이다. 쉘 프롬프트에 외부 명령을 타이핑할 때 쉘은 두 개의 프로세스를 만들기 위해 fork하거나 그 자신을 복제하고, 그런 다음 두 번째 프로세스(자식 프로세스)는 프로그램을 exec함으로써 그 자신을 다른 프로그램으로 변형한다. **exec**와 **execve**라는 두 개의 기본적인 exec 시스템 호출이 있다. 첫 번째 형태는 새로운 실행파일에 환경변수를 전달하지 않으며, 두 번째 것은 전달한다. 모든 exec들을 캡처했다는 것을 확인하기 위해 probe tuple에 둘 다를 지정할 필요가 있다.

어떻게 기술할 것인가?(만약 어떤 움직임도 보이지 않는다면 다른 윈도우 창에 몇 가지 명령을 내려 로깅을 해서 확인해보아라.)

exece 시스템 호출이 이루어질 때마다 기본 출력 결과를 보아야 할 것이다. 하지만, 이 스크립트는 어떤 프로세스가 **exec**를 하는지 말해주지 않는다. 이 정보를 표시하기 위해 우리는 내장(built-in) 변수와 표준 출력을 하는 방법이 필요하다.

액션 기록 및 내장 변수들

D 언어는 *trace()*라는 출력 액션을 제공한다. 단일 아규먼트로써 그것은 D 수식(expression)을 가진다. 추가로, D는 일련의 내장 변수들을 제공하며, 그것들 중 하나는 *execname*인데, 이것은 probe가 실행되도록 한 프로세스의 이름이다. 아래의 경우, *exec*라는 프로세스일 것이다. 다음 수정된 스크립트는 기본 출력에 덧붙여 이 정보를 보여줄 것이다.

```
syscall::exec*:entry
{
   trace(execname);
}
```

exec되는 프로그램의 이름도 역시 출력하게 하는 것은 이 스크립트를 좀더 완벽하게 만드는데 도움이 될 것이다. 각 타입의 시스템 호출은 그것에 전달되는 특정 아규먼트들을 가지고 있다. DTrace는 시스템 호출 아규먼트들을 가지고 있는 일련의 내장 변수들(*arg0 ~ arg9*)을 가지고 있다. *exec*의 경우 *arg0*은 exec되는 프로그램의 이름을 가질 것이다. 하지만, 실제로 전달되는 것은 문자열에 대한 포인터이고, 그래서 그 *arg0* 변수는 포인터로 취급되어야 한다(Solaris 레퍼런스 매뉴얼은 각 시스템 호출에 대한 아규먼트들의 목록을 제공한다).

서브루틴들 및 문자열 변수들

우리의 포인터는 한가지 문제를 가지고 있는데, 그것은 D 프로그램이 실행되고 있는 같은 주소 공간[9]에 있지 않은 문자열을 가리키고 있다. 그 D 프로그램은 커널에서 실행되고, 반면 *arg0*의 그 주소는 *exec*를 호출한 프로세스의 문자열을 가리킨다.

***DTrace hint**: 비록 **dtrace** 명령이 주소 공간에 실행된다 해도 그 D 프로그램은 DIF(D Intermediate Format)라는 형태로 컴파일 된다. Probe가 시동되면 그 probe와 관련된 어떤 DIF 프로그램들도 커널에서*

[9] 주소 공간은 거리 주소처럼 많은 일을 한다. 당신이 201번지를 찾는다고 가정해보자. 201 Main Street와 201 Oak Street가 있지만 같은 위치에 있지 않다. 그래서 당신의 probe 구문이 그것의 아규먼트들의 하나로 포인터를 받는다면 그 주소는 그 probe가 시동되도록 한 프로세스의 환경 속에 있을 것이다. 그것은 201번지일 수 있으나 당신의 거리에 있는 주소가 아니다.

*실행된다. 이것이 그 프로그램이 arg0에 있는 포인터로 직접 접근하지 못하는 이유이다. (DIF 프로그램은 버퍼에 데이터를 기록한다. 나중에, **dtrace** 명령을 통해 그 버퍼로부터 기록된 그 데이터를 가져온다.)*

userland로부터 문자열에 접근하기 위해 우리를 위해 데이터를 가져다 줄 어떤 내장된 함수가 필요하다. DTrace는 제한적이기는 하지만 한 세트의 내장 함수들을 가지고 있다. 이 함수들 중 하나인 **copyinstr**는 아규먼트로써 포인터를 가지며, 다른 주소 공간으로부터 null로 종료되는 문자열을 리턴해준다. 그것은 probe 구문 스크래치 메모리(probe clause scratch memory)에 그 문자열을 복사함으로써 이 작업을 하는데, probe 구문이 끝날 때 자동적으로 교정(reclaim)된다.

하지만 *copyinstr*는 문자들의 배열에 포인터를 단지 리턴해주지는 않는다. C와는 달리 D 언어는 실제 문자열 타입을 가지고 있다. 이것은 만약 문자열 타입의 변수를 선언한다면 어디서 이 변수를 참조하던지간에 D 프로그램은 그것을 문자열(포인터, 단일 문자, 또는 문자들의 배열이 아니라)로 사용할 것이라는 것을 의미한다. 이것은 또한 문자열들이 할당 연산자(=)를 이용해 할당될 수 있으며, 관계 연산자들을 이용해 비교될 수 있다는 것을 의미하기도 한다. 내장 변수 *execname*는 *copyinstr()*로부터 리턴된 값처럼 문자열 타입이다.

여태까지 기술한 것을 바탕으로 새로운 D 프로그램을 만들면 다음과 같다.

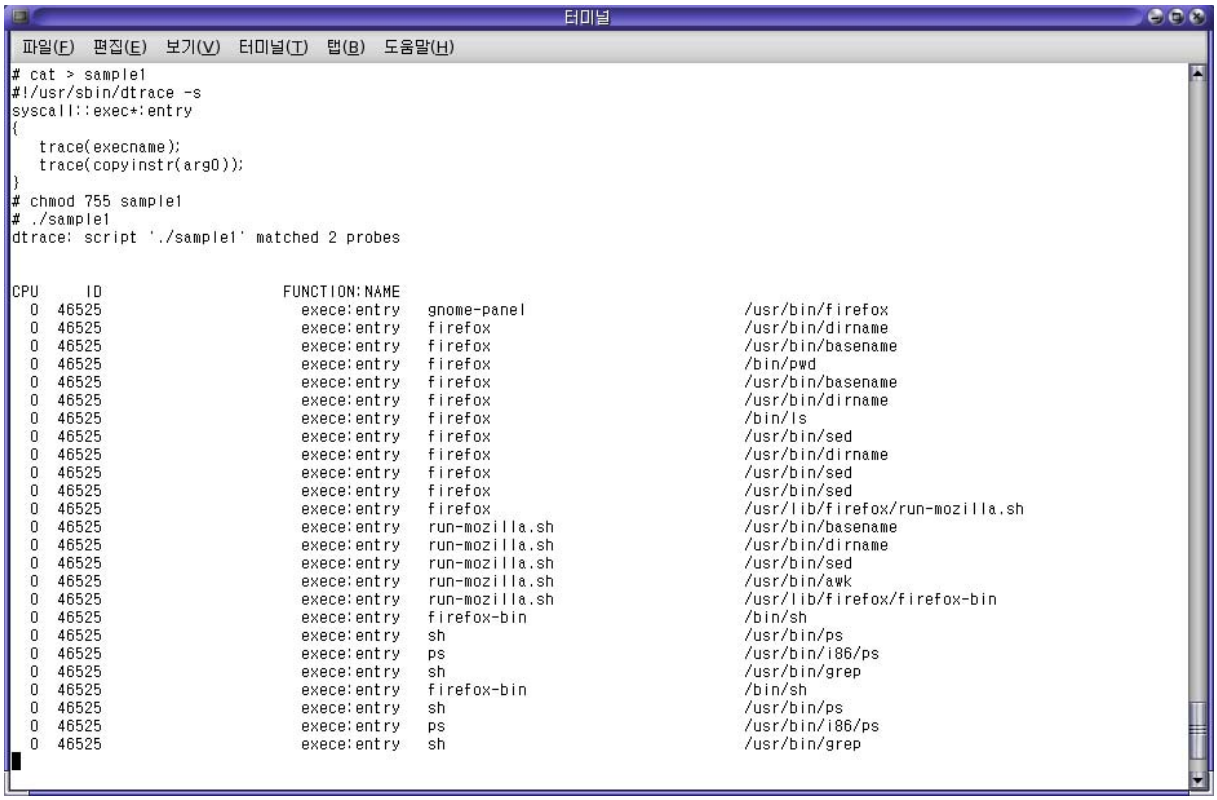
```
syscall::exec*:entry
{
   trace(execname);
   trace(copyinstr(arg0));
}
```

위 프로그램은 제대로 작동은 하겠지만 아주 깔끔하게 출력되지는 않을 것이다:

```
CPU     ID      FUNCTION:NAME
  0    106      exece:entry   ksh     /usr/bin/ls
  0    106      exece:entry   ksh     /usr/bin/who
```

~~~~~~~~~~~~~~~~~~~~~~~~ **역자의 시스템에서의 결과** ~~~~~~~~~~~~~~~~~~~~~~~~

아래는 역자의 시스템(VMware에 설치된 Solaris 10)에서 실행한 결과이다. 먼저 sapmle1이라는 스크립트를 만든 후 실행 퍼미션을 주었고, 그런 다음 파일을 실행시켰다. 스크립트를 실행시키면 "dtrace: script './sample1' matched 2 probes"라는 말만 출력된다. 이건 프로세스가 현재 실행되고 있지 않기 때문에 별다른 출력 결과가 나오지 않고 있다. 그래서 firefox를 실행시키고, 몇 번의 페이지 이동을 실행하면 다음과 같은 결과가 나온다.
~~~~~~~~~~~~~~~~~~~~~~~~



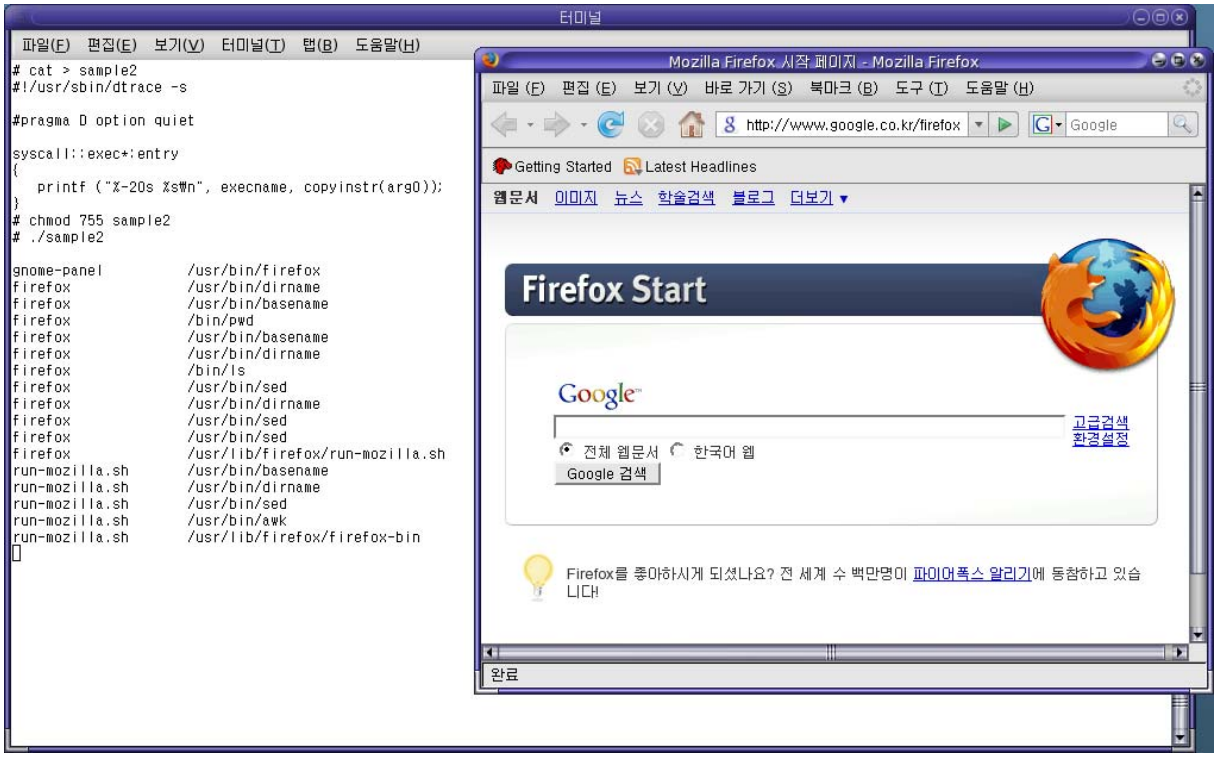
~~~~~~

좀더 보기 좋도록 하기 위해 몇 가지 기능에 대해 더 알아보자. 먼저, **trace** 액션을 사용하는 것 대신 우리는 **printf**를 사용할 것이다. 대부분 D 언어의 **printf**는 C에서와 비슷하게 작동한다. 둘째, 출력 결과를 좀더 깔끔하게 만들기 위해 우리는 기본 출력을 제거할 것이다. D는 quiet 옵션을 가지고 있는데, 이것은 커맨드 라인에서 **–q** 옵션을 사용하거나 D 프로그램에서 pragma를 이용하는 것을 통해 이루어 질 수 있다. 필자는 아래 예에서 pragma를 사용했다.

```
#pragma D option quiet

syscall::exec*:entry
{
   printf ("%-20s %s\n", execname, copyinstr(arg0));
}
```

다음은 역자 시스템에서의 결과이다.
~~~~~~



Thread-Local 변수들

이제 D 프로그램을 구성하는 것은 거의 다 끝냈지만 포인터 문제가 한가지 더 남아 있다. *arg0*에 있는 주소가 실제 메모리(디스크 상의)에 있지 않은 메모리의 위치를 가리키는 가능하다. 이것은 문자열이 이전에 접근이 안되었던 상수이거나 또는 그 문자열이 어떤 지점에서 page out되었기 때문에 발행할 수 있었다. DTrace는 그와 같은 주소에 접근할 수 없는데, 이것은 커널에 상주하는 probe 구문이 비활성화된 인터럽트와 함께 실행되기 때문에 디스크 I/O가 완료되기를 기다릴 수 없기 때문이다. 이 트릭은 시스템 호출이 완료되기까지 접근을 연기하는 것이 필요하다. 이 테크닉은 exec 시스템 호출이 리턴할 때 실행되는 또 다른 하나의 probe를 가지고 있을 필요가 있다.

하지만, *return* probe가 실행될 때 *arg0*은 다른 값을 가질 것이다. *entry* probe로부터 *arg0*에 있는 그 주소로 우리는 무엇을 할 것인가? 우리는 arg0을 저장할 필요가 있으며, 그래서 우리는 나중에 *return* probe 구문에 있는 주소를 참조할 수 있다. 하지만 어디에 그것을 저장해야 하는가? Solaris가 멀티 쓰레드 운영체제이기 때문에 동시에 많은 exec 시스템 호출이 진행될 수 있다. 우리는 각 쓰레드에 대해 각각의 저장 위치를 허용하는 방식으로 포인터를 저장할 필요가 있다.

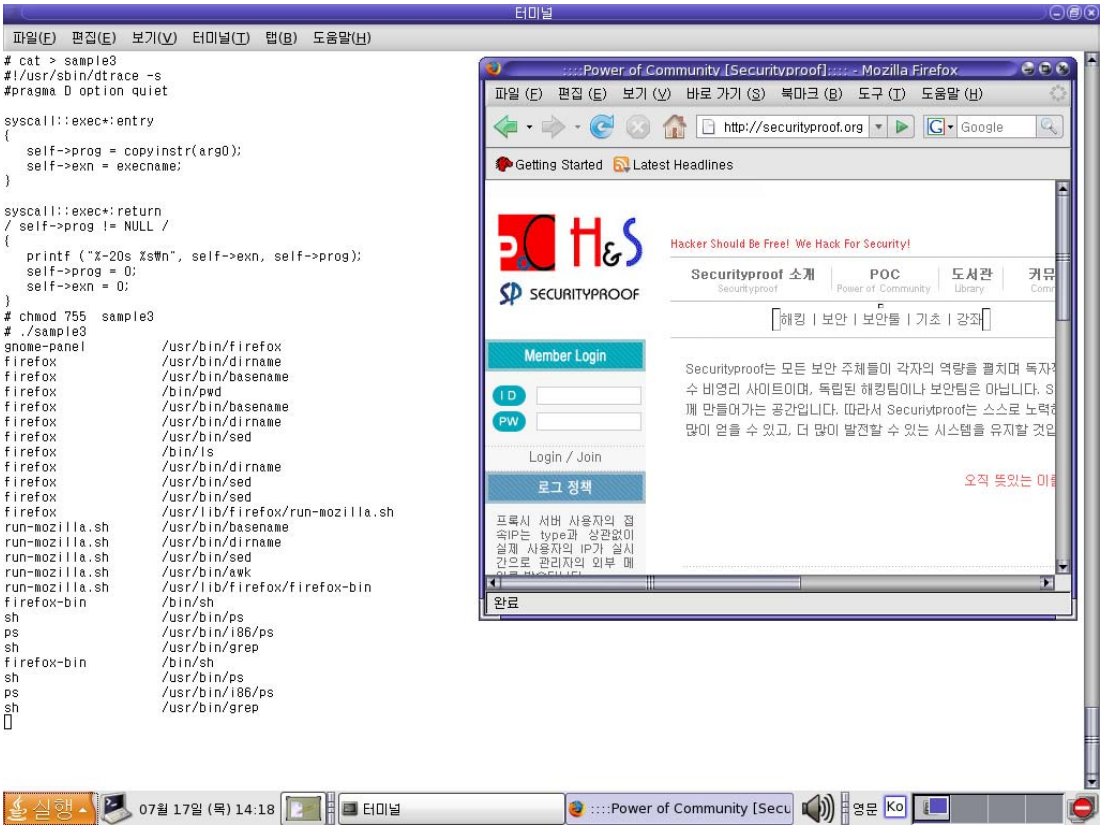
DTrace는 *thread-local*이라는 변수 타입을 가지고 있다. *thread-local*라는 변수가 선언될 때 그 probe를 실행하는 새로운 쓰레드에 대해 복사본이 만들어진다. 이를 참조하기 위해 그 변수 이름 앞에 *self->*를 위치시키면 된다. 그래서, *exec:entry* probe를 실행시키고 그럼 다음 *exec:return*

probe를 나중에 실행시키는 하나의 쓰레드는 같은 thread-local 변수들에 접근하는 것이다. 하나의 DTrace가 실행되는 동안 아주 많은 쓰레드가 있을 수 있으며, 그래서 더 이상 필요하지 않는 변수들을 제거하는 방법이 있어야 한다. thread-local 변수에 0을 할당하는 것은 그것의 스토리지를 de-allocate한다. 최종 스크립트는 다음과 같다.

```
#pragma D option quiet

syscall::exec*:entry
{
   self->prog = copyinstr(arg0);
   self->exn = execname;
}

syscall::exec*:return
/ self->prog != NULL /
{
   printf ("%-20s %s\n", self->exn, self->prog);
   self->prog = 0;
   self->exn = 0;
}
```



DTrace는 할당문이나 또는 명백하게 하나의 선언으로부터 수반된 thread-local 변수 타입을 갖는다. 만약 그 타입이 포함되면 그 할당문은 D 코드에서 그 변수를 사용하기 전에 있어야 한다. 하지만 만약 probe가 초기화되기 전에 thread-local 변수를 참조하는 순서로 사동될 경우 그 참조는 0을 리턴할 것이다.

그렇다면 왜 우리는 **self->prog**가 두 번째 구문의 predicate에서 설정되었는지 점검해야 하는가? 만약 이 predicate를 가지고 있지 않은데 exec 호출이 진행 중인 순간에 우연히 DTrace 프로그램을 실행시킨다면 어떤 일이 발생할까? 그 쓰레드에 대해 실행할 첫 번째 probe는 return probe가 될 것이다. Entry probe가 아직 실행되지 않았기 때문에 thread-local 변수들은 초기화되지 않았고, 출력 결과도 잘못된 결과들(이 경우 0들)을 포함할 것이다. 이런 타입(어떤 probe가 먼저 실행되지 확실하게 모르는)의 체크는 DTrace에세 일반적인 구조이다.

출력 결과는 다음과 같아야 할 것이다.

```
sh              /usr/bin/ls
sh              /usr/bin/csh
csh             /usr/bin/pwd
```

주어진 문제를 해결하는 데는 보통 한가지 이상의 방법이 있다. 다음은 거의 같은 출력을 해주는 대체 방법이다.

```
#pragma D option quiet

syscall::exec*:entry
{
   self->exn = execname;
}

syscall::exec*:return
/ self->exn != NULL /
{
   printf ("%-20s %s\n", self->exn, execname);
   self->exn = 0;
}
```

*exec*로부터 리턴하자마자 *execname*는 exec되고 있던 프로그램의 이름을 가지고 있고, 그래서 entry probe에 *arg0*를 저장할 필요가 없다. 하지만 만약 이 스크립트를 실행해보면 그 출력물이 첫 번째 스크립트의 출력물과 약간 다르다는 것을 보게 될 것이다.

요약

이 두 개의 글에서 필자는 DTrace의 가능성과 D 언어의 표면만을 다루었다. 앞으로 필자는 다른 probe provider[10] 들을 다룰 것이며, 여러분들에게 집합체(aggregation)라는 아주 유용한 통계 기능을 소개할 것이다. 궁극적으로 우리는 시스템 문제들을 해결하기 위해 DTrace를 사용하는 몇 가지 방법들을 살펴볼 것이다. 그동안 만약 여러분들이 기다릴 수 없거나 내장된 변수들, 서브루틴, probe provider 등에 대한 정보를 더 원한다면 Solaris Dynamic Tracing Guide[11] 를 참고하길 바란다.

*Chip Bennett (**cbennett@laurustech.com**) is a systems engineer for Laurus Technologies (**http://www.laurustech.com**), a Sun Microsystems partner. He has more than 20 years experience with UNIX systems, and holds a B.S. and M.S. in Computer Science. Chip consults with Laurus and Sun customers on a variety of Solaris related topics, and he co-chairs OS-GLUG, the OpenSolaris Great Lakes User Group (Chicago).*

[10] (역자 주) probe provider는 특정 타입의 probe를 삽입('instrumentation')을 실행하는 커널 프로그램

[11] http://docs.sun.com/app/docs/doc/817-6223