



Introduction to ProbeVue

NY Metro PowerAIX/Linux User Group Meeting
June 24th 2010

William Favorite
The Ergonomic Group

Let me start by emphatically stating that I am a reluctant PowerPoint user for presentations. My strong preference is to present directly with only occasional use of a white board. Because it was necessary to present chunks of code at this presentation I have broken down and chosen to use the “industry standard” tool for presentations. So, first, please forgive my inept attempts at using this tool. Second, my hope is that if/when this document is published online it can (to some extent) stand alone without my oral presentation.

The belief is that a presentation like this will simply carry too much info for the allotted time so the hope is that having a stand-alone presentation will serve as a means to refresh the viewer on the subject when viewing it days, weeks, or months after the presentation.

Furthermore, the presentation must be restricted for time, audience participation, and technical capabilities of those in attendance. So I have included slides with the possibility that they can be skipped for these reasons.

NOTE: Be careful when copying code examples out of PowerPoint. I have noticed that they have a tendency to pick up unwanted & invisible characters when copied *out*. I assure you, they all worked when they went *in*.

Describing ProbeVue

It can be difficult for the new user to approach a tool as radically different as a lightweight dynamic trace utility. The first step is understanding what it is.

ProbeVue is:

- a lightweight dynamic trace utility
- a customizable environment that runs Vue code
- a single source for performance and system event information
- potentially difficult for those not knowing C
- a tool to define probes, the data to be collected, how to analyze it, and how to represent it
- the future of system introspection

Admittedly, this slide is somewhat forward looking. The important truth of the matter is that there is a significant effort on the part of all dynamic trace tool creators to improve the number of providers and ultimately the number of probe points. As this number grows, the dynamic tracing tool becomes more and more a focus of those looking to monitor a Unix system.

By comparison, the approach taken by Sun with DTrace is to make it a single point for all performance event related data. To be true, this vision is short of reality as well, but the approach is one that makes these tools much more usable as data can be collected and managed through one tool. This central focus allows for two key concepts; first, that performance events can be closely correlated and second, that the administrator does not need to move from one tool to another as she diagnoses a problem.

How is ProbeVue Different?

What makes a dynamic trace utility unique
from other Unix tools?

- Filtering of probed events is easier and much more flexible than other tools. Vue allows you to filter for specific follow-on events based on previous data. For example, I only want to monitor I/O to files that were opened with a specific characteristic.
- Vue developers can select exactly what data they wish to collect about an event. This can be as simple as counting an instance of an event in a single variable or capturing the equivalent level of single event data as trace, truss, or related tools.
- Data collection can be managed in “real” time from sources instead of parsing the output of a trace log / report.

Event filtering can happen through a predicate clause that determines if an action block should execute (at runtime) or by limiting the number of registered probes (at compile time). Limiting the number of registered probes can have a much lesser effect on the ultimate load of a trace tool (this is the lightweight aspect). To the user this is a very subtle distinction as programmatically these seem very similar. In both cases it offers a flexibility (the dynamic aspect) that previous trace tools did not.

The benefit of parsing data in (near) real time is that utilities such as truss and trace can only capture a limited amount of data because they do not know what part is of value for future analysis. With first hand access to the data in real time the possibilities open up to searching through larger data sets for specific data. To do this after the fact with trace/truss the entire data set of the parameters to a call like read() or write() would need to be captured.

The passwd snoop utility (AKA: The “A Nefarious Vue Example” slide) in this presentation would normally capture read operations for every library and message file loaded. The script presented here can filter on exact file descriptors and even on size and content of the read() or write() operation.

Comparisons to Similar Tools

	probevue	trace	truss	topas(ish)	Profiler / Debugger
Provider	Various (modular)	kernel-trace(d)	/proc	PerfStat	wrapper (varies by tool)
Focus	System & App	System	App (at System)	System	App
Operation	Dynamic simultaneous capture/analyze/report	Capture, then extract, analyze, and report	Continuous capture - report	Query, analyze, report	App is typically launched from profiler tool
Gestalt	Language / scripting environment	Wide ranging tool with deep and granular insight	Focused tool with highly selectable capture	Generally single use tool restricted by design	Focused tool for app developers for code optimization
Insight	Deep & wide but limited	Exceptionally deep and wide	System calls and signals by PID	PerfLib provides virtually all system metrics	Deep into userspace code only
Strength	Versatility	Depth of insight	Simplicity	lightweight	Depth of targeted insight
Weakness	Limited insight	System load, complexity	Limited scope	limited app insight	Specialized install, limited scope

ProbeVue While still young, and lacking a depth of providers / probe points, ProbeVue offers distinctly unique capabilities from other tools. ProbeVue uses multiple providers for probes. For example, ProbeVue can use trace hooks as probe points, and it relies upon /proc for some capabilities.

trace The trace binary name is a bit misleading as there are multiple “wrapper” utilities that capture a set number of hooks, and then parse/format the results. The key difference between trace and any of these other tools (after system load) is the complexity of rolling your own utility using this tool. Even finding the right hooks can be challenging for most admins, let alone retrieving the data from a trace file, parsing the binary data, and then displaying it. The PerfStat API looks absolutely *simple* by comparison.

truss and all proc based tools are focused on looking at the interface between the application and the system. The interface being system calls, faults, and signals. (truss can view library calls, but tends to be used for the system interface.) The phrase “application” here referring to a specific application being watched.

topas is really a reference to all LibPerfStat based tools. I chose topas as the header for this column because so many of the LibPerfStat APIs are *clearly* working under the covers. ProbeVue does not have a LibPerfStat provider at this time, and there does not appear to be a linkage on the roadmap – although I think there should be.

Profiler is really a generic reference to any profiler packaged with a development environment (possibly excluding Java tools). The key point is that profilers tend to be geared to a specific binary type and do not come with the system. So for an administrator to profile an application on a production system, at least some of the development tools must be installed in the production environment, the application may require a special debug build, and it is likely that the application will need to be launched from the profiler process.

Where ProbeVue Fits

What makes a dynamic trace utility of value to you? The following list is a very limited sampling of what Vue can do for you. With the proper effort the possibilities are extensive.

- Time slow `write()`s and `read()`s for a process
- Who is making excessive syscalls on the system
- Watching access to a file
- Snooping on internal C/C++ calls within the application – without a developer profiling tool
- When a single syscall is made by anyone (like `reboot()`)
- Getting very specific I/O details for a single process
- I/O snoop – Watching writes on the `passwd` utility

This slide is self-explanatory – even though the examples are kind of nutty. The point is to relate to the reader the many possibilities of a dynamic tool. This list is really bounded only by imagination.

A side effect of this slide is the point that single purpose tools (`vmstat`, `iostat`, etc..) tend to be more limiting than environments. A holistic tool that mixes these capabilities can see across multiple areas and have a more meaningful vision into the system.

A Nefarious Vue Example

```
#!/bin/probevue

__thread char *bufptr;
__thread int flag;

int read(int fd, char *buf, unsigned long size);
int write(int fd, char *buf, unsigned long size);

@BEGIN
{
    printf("-----\n");
}

@syscall:"read:entry"
when((__pname == "passwd") && (__arg1 == 5))
{
    thread:bufptr = __arg2;
    thread:flag = 1;
}

@syscall:"read:exit"
when((thread:flag == 1) && (__rv == 1))
{
    String readbuffer[8];

    readbuffer = get_userstring(thread:bufptr, __rv);

    printf("%s", readbuffer);
}

/* Continued */

@syscall:"read:exit"
when((thread:flag == 1) && (__rv == 1))
{
    String readbuffer[8];

    thread:flag = 0;

    readbuffer = get_userstring(thread:bufptr, __rv);

    printf("%s", readbuffer);
}

@syscall:"write:entry"
when((__pname == "passwd") && (__arg1 == 1))
{
    String writebuffer[64];

    writebuffer = get_userstring(__arg2, -1);

    if(strstr(writebuffer, "Changing"))
        printf("%s", writebuffer);
}

@syscall:"exit:entry"
when(__pname == "passwd")
{
    printf("-----\n");
}
```

This slide is used twice. Here it is used to present the level of logic used and to (somewhat) sensationalize ProbeVue's capabilities as a continuation of the previous slide's appeal to the reader.

A more detailed explanation comes with the later slide, but this is a script that snoops all important I/O on the passwd utility. As a result it prints user names and passwords as modified by the passwd utility.

Understanding Vue

A quick overview of the Vue language.

- While Vue is largely C-based, Vue is an event driven language, not an imperative language like C. Functions and complex flow control is not supported.
- Vue is structured similar to awk scripts.
- Action block code and supported variables are a subset of C, yet simultaneously offer operators more common in C++/Java.
- Elements of shell such as reading command line parameters, environmental variables, and string handling.
- Additional complex data types such as associative arrays, strings, lists, and high resolution timestamps

Much of the IBM documentation refers to Vue as being “awk-like”. I think the comparison is good in that 1) they are written to respond to situations as they are encountered, 2) they have similar structure. The comparisons soon fall apart after that as the problem domain and language elements are entirely different.

An “event driven” language is where you define what to do when an event happens. This differs from an imperative language (a loose classification) in that an event driven program does not (necessarily) have a clear flow. Imperative languages move to a specific end possibly assisted by user input. The flow of a Vue “application” is dependent on events that happen (typically) outside the ProbeVue environment and (in most cases) do not have a predictable sequence of events.

The key skills that make this “language” / environment approachable are:

- Knowledge of the C language
- Knowledge of system calls
- Knowledge of shell scripting and some high level language experience such as Perl, Java, or C++.

Foreign Concepts for the First Time User

When first approaching a dynamic tracing utility there are a few concepts that are confusing to the new user.

- Vue is event driven, the language does not have a flow. (It does not read like regular code – it reads more like a library or a class with `main()` existing somewhere else.)
- The `libc` calls are not what gets called at the system (kernel) layer. (There is no `gethostbyname()` system call.)
- What does trace mean? Tentative tracing, trace buffers, `trace()` API, and the trace provider are all kind of confusing.
- What is a thread local variable? How is it different than a local (aka: automatic) or a global?
- The memory you want is not yours. You need to make a local copy to actually use it.

These are some of the concepts that first confused me when looking at a dynamic trace environment. This is really a continuation of the previous slide, but with a focus on what hurt *my* brain the most when I first looked at dynamic tracing.

The event driven nature of the language is not too hard to grasp, it just makes the code seem awkward at first.

Understanding that `libc` calls are not syscalls was one of the first problems I ran into when attempting to write a script. The `syscallx` provider in Vue and the `-l (ell)` option to `dtrace` can be useful in determining what is a syscall and what can be probed. The `gethostbyname()` example is a `libc` call that can be probed by the UFT provider – but UFT probes are not system wide, and it is not used by all applications that look up names.

The different “meanings” of the word “trace” can be confusing. Here is a simple glossary:
`trace()` – A raw “dump” of a variable passed as a parameter. This is an API that takes a data item as a parameter. `printf()` is a close equivalent.

Tentative tracing – The ability to send data to output (without committing it) and then, later, actually printing it or discarding it.

trace buffers – The output of the `probevue` command. A close analogy would be the `stdout` buffer. Unlike `stdout`, trace buffers do not “block” and can be overrun with excessive data.

trace provider – This is a provider that is actually the legacy trace command. It uses the same hooks from trace but they are available within Vue as a probe point.

There are examples of a thread local variable here in this presentation.

Memory (pointers) that you get from a syscall (or similar provider) are pointers that are relevant in *another* process. The memory must be paged in, and only then can it be copied into the `probevue` environment. Once it is in the local `probevue` environment, it can be searched, managed, printed, etc...

Sample Vue Script

```
#!/bin/probevue

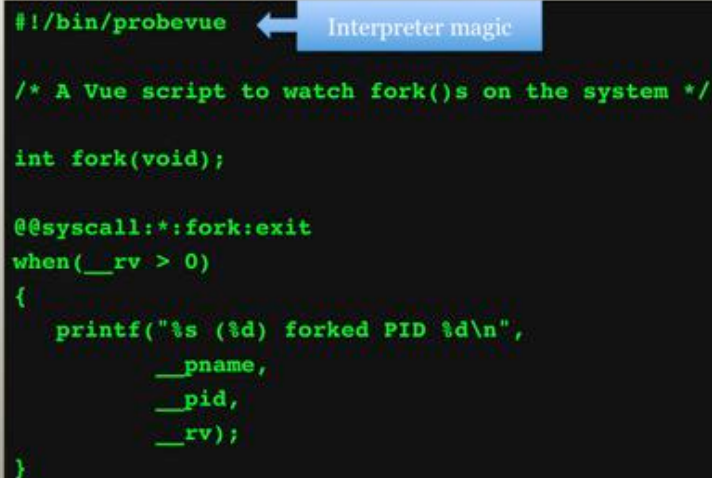
/* A Vue script to watch fork()s on the system */

int fork(void);

@@syscall:*:fork:exit
when(__rv > 0)
{
    printf("%s (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```

This is a simplistic, yet runnable script. Notes on specific sections will be distributed across the following (transitional) slides.

Sample Vue Script

A screenshot of a code editor showing a Vue script. The first line is `#!/bin/probevue`. A blue rectangular callout with the text "Interpreter magic" and a white arrow points to this line. The rest of the script is a C-like code block that uses the `@syscall` directive to watch for `fork` system calls and prints the process name, PID, and return value.

```
#!/bin/probevue  
  
/* A Vue script to watch fork()s on the system */  
  
int fork(void);  
  
@@syscall:*.fork:exit  
when(__rv > 0)  
{  
    printf("%s (%d) forked PID %d\n",  
        __pname,  
        __pid,  
        __rv);  
}
```

This example is of a Vue script set executable and utilizing an interpreter “magic” string at the top of the file. Vue scripts can be passed as parameters to the `probevue` binary or wrapped in a shell script (where they are passed to the `probevue` binary).


A limited set of parameters can be passed to the interpreter through the “shebang” line. When passing complex parameters, strings, or include (.i) files it is recommended that the Vue script be wrapped in a shell script. Wrapping within a shell gives additional flexibility in parameter parsing and validation.

When passing strings in a Vue script that has been set executable, the strings must be passed with escaped quotes – otherwise the strings are parsed by ProbeVue as objects (such as a variable name). This is inconsistent with most other Unix / shell tools, so the recommendation is to wrap scripts requiring string parameters in a shell script.

Sample Vue Script

```
#!/bin/probevue

/* A Vue script to watch fork()s on the system */
i 0;
@@syscall:*.fork:exit
when(__rv > 0)
{
    printf("%s (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```



Vue uses C-style comments.

C++ comments also work but are not documented, so reliance upon them is not advised. (They work probably because of the potential inclusion of C++ header files that may have C++ style comments.)

The shell interpreter line works like it does in shell, perl, and other interpreters. For this reason the # character works like a comment. It is strongly recommended that shell-style comments NOT be used. C style defines (pre-processor directives) are ignored but could conceivably be used in future implementations of Vue.

Sample Vue Script

```
#!/bin/probevue

/* A Vue script to watch fork()s on the system */

int fork(void); ← Function prototype

@@syscall:*.fork:exit
when(__rv > 0)
{
    printf("%s (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```

In order to access parameters or return values from system calls, the function must be prototyped. This implementation detail is somewhat unique to Vue amongst the dynamic trace tools on the market. These prototypes can be collected in a common include file. (Note: ProbeVue cannot parse all the complexities of a common .h file. These must be converted/simplified and usually have a .i extension in this format.) It has been suggested that a common include file be created and auto-included for each provider requiring one (uft excluded of course).

In this example we are interested in the return value from fork(). It has three possible values: -1 on error, 0 if we are the child, and >0 (the PID of the child) if we are the parent.

Note the read() prototype in the data type / context slide. It mentions some usage notes on what data type is specified when prototyping and the impact on actual data types Vue uses.

Sample Vue Script

```
#!/bin/probevue

/* A Vue script to watch fork()s on the system */

int fork(void);

@@syscall:*.fork:exit ← Probe point specification
when(__rv > 0)
{
    printf("%s (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```

This line is where all the probe points we wish to register are specified. This (probe point) specification is for the action block that follows. It is possible to have more than one probe point specification (with accompanying action block). It is also possible to specify more than one probe point in a single probe point specification. Some providers allow for wild carding (The “*” here specifies that we want to look at ALL PIDs).

The probe point is defined by:

@@**syscall**:*.fork:exit → This specifies the syscall provider

@@syscall:*.fork:exit → (For the syscall provider) the PID of a process can be specified here

@@syscall:*.**fork**:exit → The specific syscall to probe. This may map to a slightly different syscall (find actual name with get_function()).

@@syscall:*.fork:**exit** → This can be entry or exit. fork() is more interesting to us on exit as we can then determine parent-child relationship.

Sample Vue Script

```
#!/bin/probevue

/* A Vue script to watch fork()s on the system */

int fork(void);

@@syscall:*.fork:exit
when(__rv > 0) ← Predicate
{
    printf("%s (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```

The predicate clause is a limiting function on the number of probes that we want to instrument. In this case we only care about `fork()` calls that have a return value of greater than 0. (This happens when we are the parent and the call was successful. The return value will be the PID of the new child.)

This functionality could have been achieved using an `if` statement in the action block. The understanding is that this method is slightly more optimal – performance wise. When we can limit probe registration, such as specifying a PID in the probe definition line rather than the predicate, there can be a much more significant impact on performance.

D users will have a tendency to rely upon the predicate as this is the only place that D allows for this kind of logic. Vue provides for the capability to put a conditional (`if`) in the action block.


Sample Vue Script

```
#!/bin/probevue

/* A Vue script to watch fork()s on the system */

int fork(void);

@@syscall:*.fork:exit
when(__rv > 0)
{
    printf("Process (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```



The action block is what most resembles a C function. The primary differences are some syntactical extensions to C and that we do not have a parameter list or a return value like a C function, instead it relies entirely upon side effects.

While there is not an *explicit* parameter list like a C function, there are parameter-like variables that are available from this provider. The availability of those parameters is dependent upon the system call probed, the entry/exit probe used, and if the system call was prototyped. In this case, we only have one “parameter” from the probe, and that is the return value.

`__pid` and `__pname` are global read-only built-ins that are contextually set based upon where they are used. `__pid` is the process ID (an integer) and `__pname` is the process name (a string).

Sample Vue Script

```
#!/bin/probevue

/* A Vue script to watch fork()s on the system */

int fork(void);

@@syscall:*.fork:exit
when(__rv > 0)
{
    printf("%s (%d) forked PID %d\n",
           __pname,
           __pid,
           __rv);
}
```

API call

printf() conforms to the C version. The most significant difference is that you will NOT be warned when passing an invalid type as you would in the C version (by your compiler). The result is that if the format specifier is too small then significant digits may be ignored when printed. The alternative is to use the trace() function that does not rely upon format specifiers. (Personally I tend to use the trace() function to debug printf() issues. – In the event that I used the wrong format specifier for the datatype supplied.)

This lack of checking of format specifiers to datatypes was suggested as a bug in ProbeVue. At the very least the probevue binary should allow for a “strict” flag that will warn in situations where data may be mis-interpreted.

Sample Vue Script

This is the previous sample script running during a rsh login on the system.

```
# ./sample.e
inetd (168078) forked PID 229600
rlogind (229600) forked PID 352488
tsm (352488) forked PID 331782
ksh (352488) forked PID 331784
ksh (352488) forked PID 331786
ksh (352488) forked PID 331788
ksh (352488) forked PID 331790
```

This is the previous sample code running on a system. Adding probes of `exec()` and `exit()` calls may give some additional insight into what is happening here (if this were an actual point of interest).

I rsh logged into the system to show a more interesting thread of `fork()` calls than the shell running “ls” over and over.

It would be nice to offer a (wall clock) timestamp that could be matched with wall time for these events. ProbeVue does not currently offer this capability (other than reading the kernel time variable). There is no option within ProbeVue for conversion to localtime. Still, the ability to display a chronological view of forks (`exec()`s and `exit()`s) across the system is rather unique in system tools of this simplicity.

[Conceivably one could pass a parameter to `diff_time()` that was set to the Unix epoch (0) to get a more reliable result than relying on the kernel time variable. (Note: Some early versions of ProbeVue did not always return the proper value for time (from the kernel variable). This was based upon personal experience, and not a methodical / thorough testing of the problem).]

Variables

Understanding variable source, context, scope, and type can be confusing for the new user.

Datatype	All C types including arrays and struct(ures) are supported. (Floating point types are a slight exception.) Vue also supports the timestamp, string, & list/aggregation data types. Support for associative arrays have been announced.
Scope	Variables can be scoped much like C in that they can be global or probe local (aka: "automatic"). Vue introduces a concept of thread local variables that are local to the process probed.
Context	Some variables are only usable in a context such as a specific probe action block, and have no context in another such as an interval probe action block. Some variables change type (size) based upon the bitness of the binary that is traced. (This is a variation on the concept of scope.)
Provider	Probe providers offer variables that have type based upon the probe. The probevue environment offers shell expansion of \$ variables. (Exported) kernel variables are accessible.

It is difficult to relate variables in Vue to the first time user as they have several more dimensions than that of C. Scope can be complicated by the thread concept as well as relating context. Examples of these differences follow over the next few slides.

While Vue appears at first to be C based, Java and C++ developers may be more comfortable with some of the datatypes and (overloaded) operators – such as the ability to concatenate strings with the “+” operator.

The timestamp data type will seem familiar to those using the high resolution time functions that use the timeval structure such as gettimeofday(). The specific implementation details are not shared by IBM – but it is sufficient to say that mathematical comparisons on this data type essentially work like integers, but calculating differences on (subtracting) two values requires use of the diff_time() function.

String, struct, and any pass by reference data types must be copied into the ProbeVue environment as they cannot be accessed directly from the watched process. If the memory location is paged out the copy will fail (silently) and the default behavior of ProbeVue is to NOT cause a page fault.

It is the author’s opinion that the List data type and related functions are worthless. – This statement really depends partially upon how List is actually implemented. If all data points are retained in a dynamic list then it is not worth using. If List only stores min, max, count, total, and a running average (derived from total / count) then the List and associated API offers some (code simplification) value – provided you never intend to reset the data on intervals (as there is no reset option for a List data type).

Variable Scope Demo

```
#!/bin/probevue

int gi;
__thread int ti;

@@BEGIN
{
    auto:li = 0;
    gi = 0;
}

@@syscall:*fork:entry
{
    auto:li = 0;
    gi++; ti++; li++;

    printf("fork(G:%d, T:%d, L:%d):\n", gi, ti, li);

    if(10 == gi)
        exit();
}

@@END
{
    auto:li = 0;
    printf("exit(G:%d T:%d L:%d):\n", gi, ti, li);
}
```

```
#!/scope-sample.e
fork(G:1, T:1, L:1);
fork(G:2, T:2, L:1);
fork(G:3, T:1, L:1);
fork(G:4, T:1, L:1);
fork(G:5, T:1, L:1);
fork(G:6, T:2, L:1);
fork(G:7, T:3, L:1);
fork(G:8, T:4, L:1);
fork(G:9, T:5, L:1);
fork(G:10, T:3, L:1);
exit(G:10 T:0 L:0);
```

This is a demonstration of three different variables in a scoping exercise.

I used the fork() syscall because that is something that I can easily control on a relatively quiesced system.

What is happening in this example (Using the global counter variable):

- 1: ls in a shell
- 2: ls in a shell
- 3-9: A rsh login
- 10: ls in the original shell

The global variable increments as expected.

The thread variable increments as expected (there are multiple processes calling fork()).

The local variable is always local and never increments beyond 1. (See the NOTE below.)

The key points:

- The thread local has no context in the BEGIN or END blocks.
- The global behaves like all globals everywhere.
- Use locals with caution – they can bite their owners if not initialized (see NOTE below).

NOTE: The li (local int) is declared and initialized as such. The only reason it appears to work as a local is because it is re-initialized in each action block. It DOES NOT actually function as a local, but instead as a global. Some variations on the local usage and how it “responds” are recorded here:

- Declared using “__auto int li;” at the top of each action block → acts like a global.
- Declared and used only in syscall block (using __auto with no initialization) → acts like a global (keeps its value between runs).
- Declared and initialized in BEGIN (or syscall) and accessed elsewhere → acts like a global. It is “known” in the END block even though not declared there.

This is a TL3 system.

Initially I tried to write an example with a variable namespace clash between a local and a global (using a variable of that name). It failed on compile with the following error:

ERR-67: Line:12 Column:9 Variable 'clash' already declared at line 3.

Variable Types & Context

```
#!/bin/probevue
long glong;
int gint;
char *cp;

long write(int fd, char *s, long size);

@@BEGIN
{
    printf("global int = %d\n", sizeof(gint));
    printf("global pointer = %d\n", sizeof(cp));
    printf("global long = %d\n", sizeof(glong));
}

@@syscall:*:write:entry
{
    printf("process int = %d\n", sizeof(__arg1));
    printf("process pointer = %d\n", sizeof(__arg2));
    printf("process long = %d\n", sizeof(__arg3));
    exit();
}
```

```
# ./sizesamp.e
global int = 4
global pointer = 8
global long = 8
process int = 4
process pointer = 4
process long = 4
```

The context of a variable determines both its scope but also its data type (size).

The size of a pointer and a long vary by the bitness of the probed process, an int does not. Variable size types assume 64 bit values in the ProbeVue environment, but the first write (in this example) happened in a 32 bit process.

There is no known (to me) method for determining bitness of an application from probevue (short of calling sizeof() on a context sensitive / variable size data type).

NOTE: The prototype for write() depends on defines but usually maps to the following line inunistd.h:

```
extern ssize_t write(int, const void *, size_t);
```

ssize_t is a signed long (typedef in sys/types.h)

size_t is a long (typedef in sys/types.h)

[Personal note: My understanding of the “extern” keyword in this context is that the function is really provided by the kernel and is not actually in libc.]

The data types specified in the prototype do not matter to the results of this script. Declaring int or (unsigned) long in this example does not change the results. (Most of the examples in the Extended User Guide Specification use “int” in place of size_t.)

Kernel Variable Example

```
#!/bin/probevue

__kernel long time;  /* Seconds from epoch */
__kernel long lbolt; /* Ticks from boot    */

@@BEGIN
{
    printf("Seconds past Unix epoch  : %lld\n",
           time);
    printf("Seconds past system start: %lld\n",
           lbolt / 100);
    exit();
}
```

This is a (greatly) simplified version of accessing a pair of kernel variables.

The version that is in the Extended User's Guide Specification makes reference to more variables – of greater interest.

Kernel variables cannot be used in predicates (although you should be able to copy the value into a local and compare that way).

Providers

Providers are “modules” that offer probe points for a Vue script

syscall	Probe points for entry and exit from select system calls.
Interval	Probes that can fire on an interval. At this time an interval probe does not have process context.
Internal	Probes that fire on the start and end of a ProbeVue session typically for setup and printing.
UFT	(User Function Tracing) Probes that are defined on the entry into a user function. (C and now C++)
Trace	A limited number of trace hooks can be probed. Trace data is available in context sensitive variables and must be parsed (just like they would be from the trace file).

syscall – Covered extensively here

interval – A timer

internal – BEGIN and END probes

UFT – This is basically a “umbrella” provider for the original C uft provider and the newer C++ version. These are different providers and have different syntax.

trace – Not really covered here. Note that this is limited access to trace hooks and NOT the trace() call.

Additionally:

syscallx – REGEX capable version of syscall. It is not documented at this time, so appears to be semi-unofficial. There is a slide on this at this time.

java – It is in the TL release notes, but I have not seen documentation for it, and would probably be inclined to rely upon actual Java tools, and not ProbeVue.

Probe Definition Example

```
#!/usr/bin/probevue
long timeinsec;

@@BEGIN
{
    timeinsec = 0;
}

@@syscall:$1:exit:entry
{
    exit();
}

@@interval:*:clock:1000
{
    timeinsec++;
}

@@END
{
    printf("Elapsed time : %d seconds.\n", timeinsec);
}
```

This is a nonsensical script that watches a PID from the time the script was started until the PID exits, then reports the time. This is kind of like the “wait” command. The point here is to demonstrate several different probe definitions.

NOTE: timeinsec fell out of scope in the syscall action block when run on TL3. I initially printed elapsed time in the syscall action block to save space in the slide – but consistently got a value of 0. Explicit global declarations and references made no difference to the outcome. This problem disappeared once the system went to TL4.

Probe Definition Example

```
#!/usr/bin/probevue
long timeinsec;
@@BEGIN ← Simple BEGIN probe point
{
    timeinsec = 0;
}

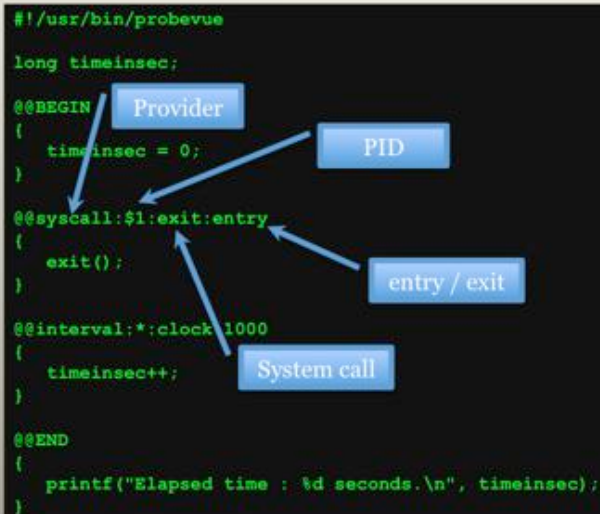
@@syscall:$1:exit:entry
{
    exit();
}

@@interval;*:clock:1000
{
    timeinsec++;
}

@@END ← END uses same format
{
    printf("Elapsed time : %d seconds.\n", timeinsec);
}
```

These are probe points provided by the ProbeVue environment. They are fired as probevue starts and as it exits. They are placed in that order here for a visual simplicity and not as a requirement.

Probe Definition Example



```
#!/usr/bin/probevue
long timeinsec;
@@BEGIN
{
    timeinsec = 0;
}
@@syscall:$1:exit:entry
{
    exit();
}
@@interval:*:clock 1000
{
    timeinsec++;
}
@@END
{
    printf("Elapsed time : %d seconds.\n", timeinsec);
}
```

The diagram shows a script with four blue boxes and arrows pointing to specific lines: 'Provider' points to '@@BEGIN', 'PID' points to 'timeinsec = 0;', 'entry / exit' points to '@@syscall:\$1:exit:entry', and 'System call' points to 'timeinsec++;'.

This is the syscall provider.

Field 1 is always “syscall” (There is a syscallx version).

Field 2 can be wildcarded for this provider. Here it is set to the first parameter to the script. If the parameter is not passed or is invalid (no explicit checking is done in the script) then the probes will fail to register as being of invalid format. So there is an “implicit” checking of this value.

Field 3 is the syscall name. As noted elsewhere (in this presentation) this cannot be wildcarded (the syscallx version supports a wildcard (*)) and is NOT the libc API but the kernel API.

Field 4 is the entry or exit point of the syscall. The return value is only available on the exit. The parameters are only available on the entry.

Probe Definition Example

```
#!/usr/bin/probevue
long timeinsec;

@@BEGIN
{
    timeinsec = 0;
}

@@syscall:1 Provider
{
    exit();
}

@@interval:*:clock:1000
{
    timeinsec++;
}

@@END
{
    printf("Elapsed time : %d seconds.\n", timeinsec);
}
```

The diagram illustrates the fields of the probe definition. It shows four blue boxes with arrows pointing to specific parts of the code: 'Provider' points to the 'Provider' field of the `@@syscall` line; 'Reserved' points to the '*' field of the `@@interval` line; 'Timer resolution' points to the '1000' field of the `@@interval` line; and 'Always "clock"' points to the 'clock' field of the `@@interval` line.

The interval provider does not (currently) set context (such as a process). For this reason it can not be used to look at a specific process on an interval. The second field is reserved for this (potential) use.

The third field is "clock" and is for wall clock time. There is currently no other option for this field.

The final field is the timer resolution. The value here is in seconds with the minimal value (or granularity) being $1/10^{\text{th}}$ of a second (or 100).

Wildcard with syscallx

The syscallx provider offers a wildcard option on the syscall field that is not offered in the syscall provider. Unfortunately it is difficult to *fully* exploit this capability with the current language features.

```
#!/bin/probevue

@@syscallx:$1:*.entry
{
    printf("%s()\n", get_function());
}

@@syscall:$1:exit:entry
{
    exit();
}
```

syscallx is not officially published (This was successfully run on a TL3 system). Without an associative array / aggregations, this wildcard capability is of little use. At best we can only gather basic statistical info (such as the number of syscalls for this PID, or time spent in each syscall, etc..). We will not be able to dump this into an aggregation and sort by the number of syscalls, or time spent in each syscall, etc... Although, it is possible we could do this with some post processing tool like perl.

syscallx does not support an actual REGEX capability in the probe point definition. I was able to do a (simplified) REGEX using the following code:

```
@@syscallx:$__CPID:*.entry
{
    funcname = get_function();

    if(strstr(funcname, "SIMPLE_PATTERN"))
        printf("%s\n", get_function());
}
```

I would argue that the power of syscallx (over syscall) is not the ability to wildcard the function name but the ability to probe significantly more syscalls than with the original provider.

Simple UFT Example

Here we are probing for a libc function. The PID is a required field on the UFT provider so we cannot probe across the system for this call.

```
struct hostent *gethostbyname(char *);

@@uft:$__CPID:libc:gethostbyname:entry
{
    printf("gethostbyname(%s);\n",
        get_userstring(__arg1, -1));
}

@@syscall:$__CPID:exit:entry
{
    exit();
}
```

Notes:

- The libc section of the probe point specification is not necessary. This can be wildcarded.
- The interpreter magic is NOT used because we relied upon \$__CPID to get our PID. This means we use -X and -A for our application.
- When using -X, the full path to the binary is required. The probeview environment does not understand \$PATH.
- The gethostbyname() is prototyped because we access one of the arguments. If we were just looking to see if it was called, then the prototype is not necessary.
- This example *cheats* a bit by not copying to an specific / explicit local string. Here we copy to the parameter list of the printf() call.

A Nefarious Vue Example

```
#!/bin/probevue

__thread char *bufptr;
__thread int flag;

int read(int fd, char *buf, unsigned long size);
int write(int fd, char *buf, unsigned long size);

@BEGIN
{
    printf("-----\n");
}

@syscall:"read:entry"
when((__pname == "passwd") && (__arg1 == 5))
{
    thread:bufptr = __arg2;
    thread:flag = 1;
}

@syscall:"read:exit"
when((thread:flag == 1) && (__rv == 1))
{
    String readbuffer[8];

    readbuffer = get_userstring(thread:bufptr, __rv);

    printf("%s", readbuffer);
}

/* Continued */

@syscall:"read:exit"
when((thread:flag == 1) && (__rv == 1))
{
    String readbuffer[8];

    thread:flag = 0;

    readbuffer = get_userstring(thread:bufptr, __rv);

    printf("%s", readbuffer);
}

@syscall:"write:entry"
when((__pname == "passwd") && (__arg1 == 1))
{
    String writebuffer[64];

    writebuffer = get_userstring(__arg2, -1);

    if(strstr(writebuffer, "Changing"))
        printf("%s", writebuffer);
}

@syscall:"exit:entry"
when((__pname == "passwd"))
{
    printf("-----\n");
}
```

It may be appropriate to skip this slide in the presentation due to size limitations, screen display, and audience familiarity with subject.

This is the slide introduced earlier, we can now talk about it. (The conversation must come from the presenter as space in the PowerPoint comments do not allow for a full discussion.)

It is a decent presentation of the following concepts (and carries some “appeal” because of the subject matter and sensational slide title):

- Thread local variables
- Copying data from the remote application space
- Prototyping system calls
- The difference between the entry and exit probes for the syscall provider

This is dumped onto two “screens” to show all the code. This is about as efficient as this can be displayed without removing or reformatting code to save space.

What this code does: It looks for reads/writes on a process called “passwd” and dumps the results to a file or stdout with some relevant formatting.

A sample run looks like this:

```
-----
Changing password for "wfavorit"
myoldpass
mynewpass
mynewpass
-----
Changing password for "wfavorit"
userpass
userpass
-----
^C
```

The first is when run by the user, the second is by root. (Fake PWs were entered, otherwise this is a cut-n-paste from a real session.)

Some notes about the code:

- The passwd binary is suid and owned by root (instead of RBAC) so we cannot use the `__uid` or `__euid` values to determine who the target user is. Neither can we pull command line info. But we know that passwd writes a string that begins with “Changing” that includes the target users name.
- We cheat in a few places because we know that file descriptor 5 is used to read user input and that the reads for that information is always of size = 1. The “cheat” here is that I used ProbeVue to find the file descriptor and then I filtered on it specifically. The approach of using a single tool to work to a specific end is central to the power of this environment.
- This is more about making a point about Vue features and capabilities, than real value. The primary reason (that is is of limited value/threat) is that it can be only run by a super user who really does not care about passwords (as a means to personally exploit the system).

Dynamic Tracing Environments Compared

This is a simplistic overview of the available dynamic tracing environments. While Vue offers some potential language enhancements, DTrace is the most mature and remains the reference implementation.

	DTrace	ProbeVue	SystemTap
Providers	Significant	Growing	Growing
Language	Minimalist	Minimalist+	Full (loops & functions)
API	Complete	Minimal	Large and growing
Platforms	Solaris, OS-X	AIX (6.1+)	Linux (Distro specific)
Environment	Runtime compile	Runtime compile	Translate, compile, load
Support	Significant	Minimal	Minimal

This is a dangerous slide in that it is somewhat subjective. Please ingest with a grain of salt.

DTrace

This is the reference standard of lightweight dynamic tracing environments. It was the first to the game, it has the longest history, the most support, and is the most powerful of the three.

ProbeVue

The presentation is about ProbeVue so I will skip the details and say that there is a significant effort from Austin to develop ProbeVue. Changes to the environment are seen on virtually every TL.

SystemTap

I am not entirely familiar with SystemTap but I have been watching from a distance for some time.

SystemTap is the Linux (GNU*) variant. GNU is mentioned not simply out of respect to RMS, but more of an acknowledgement of licensing incompatibilities between the Sun open license on D and that of the GNU license of Linux.

The most interesting things of note on SystemTap is the metamorphic and full feature design. I see the design effects of multiple development groups pulling in different directions as well as the kitchen sink approach to features. Neither of these is meant as a compliment. The varied focus in the design means additional complexity. The full language approach means that SystemTap is capable of infinite loops (that are now modules in your kernel).

Both D and Vue seem to have a well focused / mature design that SystemTap lacks.

Others

HP is touting Ktrace and Caliper as equivalents. (Ktrace is an AIX trace “equivalent” and Caliper is a profiler.) While they are powerful tools, they are not lightweight dynamic trace tools. Furthermore (by HPs own documentation ktrace is not recommended in production environments nor can it be used in Integrity VMs.)

Also note: Because of the nature of the probes, dynamic tracing scripts are typically NOT portable across different versions of Unix (ie: Solaris to OS-X). There will be subtle variations in the system calls that will cause a script to fail on a different Unix. There is no method to move across language environments (ie: D to Vue). Although knowledge of one language translates well into the other.

The Future of ProbeVue

Because these are forward looking statements they refer to very general concepts or updates that have shipped in 6.1 TL's. The point is that IBM is actively developing this environment.

- New providers coming on line; specifically (compiler independent) C++
- Language enhancements
- API enhancements
- ToolKit / GUI

I am careful not to mention items that IBM has not committed to. This slide is generic enough and tends to reference items that are already known (and searchable on Google).

My personal requests (on top of the ones that are in the pipeline) are:

- Access to LibPerfStat: This should be fairly easy to implement as a thin logic / mapping layer that would allow a PerfStat API to be called from within the ProbeVue environment. The ability to access this significant body of metrics would be a great addition to the capabilities of the tool.
- Access to a process environment info: Access to environmental variables, and contextual info such as CWD would be great.

Tidbits – Items Not Covered

These items were largely omitted (primarily for time constraints), and can be found on the web.

- Tentative tracing – The ability to “queue” output for the trace buffer, but actually commit or disregard later.
- Shell Wrapper – Wrapping a Vue script in shell is useful for parameter validation, passing options to the binary, and avoiding annoying string escape sequences.
- Calculating percentages (floating point “emulation”).
- The List datatype – When and how to use, and when to not use the List type.

Useful URLs:

<http://www.ibm.com/developerworks/aix/library/au-probevue/>
<http://en.wikipedia.org/wiki/ProbeVue>
<http://www.tablespace.net/probevue/index.html>
<http://www.tablespace.net/quicksheet/vue-quicksheet.pdf>
<http://www.ibm.com/developerworks/aix/library/au-probevuec/index.html>
<http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds4/probevue.htm>

Personal Observations

- Solaris admins who are familiar with D/DTrace will find Vue a bit limiting. Fortunately this situation is changing with each release.
- When writing complex scripts it is best to write more modular and test each piece. It can be frustrating to write a lengthy script and find a bug in the language / environment that invalidates the entire project.
- TL 4 and later is better. I have personally created several PMRs on TL3 and earlier. This presentation was written primarily on a TL3 system. Some of the notes will reflect different results on TL3 to TL4.
- Tools have users, environments have communities. IBM does not do “community” well.

MetaData

William Favorite

URL: <http://www.tablespace.net>

Email: wfavorite@tablespace.net

Phone: 720-289-8142

A copy of this presentation can be found at:
<http://www.tablespace.net/probevue/>

Version: 0.10.0
Date: 6/22/10



Personal & presentation info.