Architecture of systemtap: a Linux trace/probe tool

Frank Ch. Eigler <fche@redhat.com>
Vara Prasad <prasadav@us.ibm.com>
Will Cohen <wcohen@redhat.com>
Hien Nguyen <hien@us.ibm.com>
Martin Hunt <hunt@redhat.com>
Jim Keniston <jkenisto@us.ibm.com>
Brad Chen <brad.chen@intel.com>

July 7, 2005

1 Motivation

A tracing and probing tool gives knowledgeable users a deep insight into what is going on inside the operating system, going well beyond isolated tools like `netstat`, `ps`, `top`, `iostat`.

1.1 Requirements

Systemtap is designed to strike a useful balance between several requirements.

Ease of use
The tool's probe language should be simple and compact. The output should be available in multiple formats. Users should be able to reuse general scripts written by others. See section 3 for a description of our scripting language.

Extensibility
The tool should allow subsystem experts to provide extensions that expose interesting data in those subsystems safely. The tool should deal with the constant drift of kernel versions. See section 4 for "tapsets", our extensibility construct.

Performance
The probes should execute fast enough that users are not discouraged from their liberal use in a live system. It should be efficient on multiprocessor systems.

Transparency
It should be possible for an expert to see details of the tool's operation, so they can convince themselves of its safety, accuracy. The tool should itself be free software, and its intermediate outputs should be potentially visible.

Simplicity
The tool must not take too long to develop, document, and deploy.

Flexibility
The tool should run on a spectrum of processor architectures and kernel versions. Both kernel and user space programs should be instrumentable, even in the absence of source code.

Safety
It should live within the many constraints of operation within the kernel. It should prevent unintentional interference. See section 5 for our treatment of this issue.

2 Systemtap processing steps

Systemtap is structured in a straightforward pipeline shown in figure 1. The steps are detailed below.

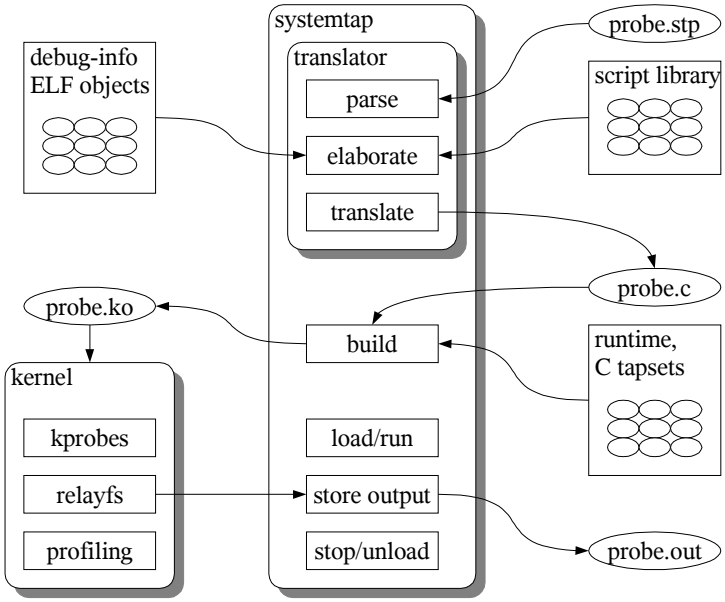

Figure 1: Systemtap processing steps

2.1 Probe language

The systemtap input consists of a script, written in a simple language. The language describes an association of handler subroutines with probe points. Probe points are abstract names given to identify a particular place in kernel/user code, or a particular event (timers, counters) that may occur at any time. Handlers are subroutines written in the script language, which are run whenever the probe points are hit. Probe points correspond to `gdb` *breakpoints*, and handlers to their *command lists*.

The language resembles `dtrace's` "D", itself inspired by the old UNIX tool `awk`. These are simplified from C, lacking types, declarations, and most indirection, but adding associative arrays and simplified string processing. The language includes some extensions to interoperate with the target software being instrumented, in order to refer to its data and program state. Section 3 describes the language in more detail.

2.2 Elaboration

Elaboration is a processing phase that analyzes the input script, and resolves any needed symbolic references to the kernel, user programs, or other any "tapsets". Tapsets are libraries of script or C code used to extend the capability of a basic script, and are described in section 4. Elaboration is analogous to linking an object file with needed libraries, turn them into a self-contained executable.

References to kernel data such as function parameters, local and global variables, functions, source locations, all need to be resolved to actual run-time addresses. This is most rigorously done by processing the DWARF debugging information emitted by the compiler, in the same way as an ordinary debugger would. However,

such debug data processing is transformed into an executable form ahead of time, so that during actual probe execution, no explicit decoding is necessary.

Debugging data contains enough information to locate inlined copies of functions (very common in the Linux kernel), local variables, types, and declarations beyond what are ordinarily exported to kernel modules. It enables placement of probe points into the interior of functions. Systemtap will exploit this extra access, which is simply not possible for a proprietary target software that omits debug data.

2.3 Translation

Once an entire set of probe functions is processed through the elaboration stage, they are translated to a quantity of C code.

Each systemtap construct is expanded to a block of C that includes whatever locking and safety checks are necessary. Control-flow constructs translate to include runaway-prevention logic. Each variable shared amongst probes is mapped to an appropriate static declaration, and accesses are protected by locks. Each group of local variables is placed into a synthetic call frame structure that keeps them off the tiny real kernel stacks.

Probe handlers are wrapped by an interface function which uses whatever probe point registration API is appropriate. For location type probe points targeting the kernel, this generally uses `kprobes`. Where the target software is user-level, probe points would need to be inserted into specific processes' executable segments, using a mechanism yet to be specified. (The existing dprobes inode-specific probe points are not a perfect match for the sort of per-user instrumentation we envison.)

The generated code includes a references to a common runtime that provides routines for generic lookup tables, constrained memory management, startup, shutdown, and I/O, and other functions.

When complete, the generated C code is compiled, and linked with the runtime, into a stand-alone kernel module. For security reasons, the module may be cryptographically signed, so that it may be archived and later reused here, or on another computer without a compiler installed.

2.4 Execution

To run the probes, the systemtap driver program simply loads the kernel module using `insmod`. The module will initialize itself, insert the probes, then sit back and let the probe handlers be triggered by the system to collect and pass data. It will eventually remove the probes at unload time.

When a probe is hit, the associated handler routine takes over the processor, suspending the target software briefly. When all handlers for that probe point have been executed, the target program resumes. Because of the suspension, handlers must not block, except very briefly on each other. They should hold only locks while manipulating shared systemtap variables, or accessing previously unlocked target-side data. On the other hand, it is necessary to hold *no* locks while calling non-user-context kernel functions like `copy_from_user`.

The probe run concludes when the user sends an interrupt to the driver, or when the probe script runs an exit primitive. (This primitive might simply send a SIGINT to the running user-level driver process.)

3 Programming

A systemtap script file has the suffix ".stp". A script file is a sequence of top-level constructs, of which there are three types: probe definitions, auxiliary function definitions, and global variable declarations. These may

occur in any order, and forward references are permitted.

A probe definition identifies one or more probe points, and a body of code to execute when any of them is hit. Multiple probe handlers may execute concurrently on a multiprocessor. Multiple probe definitions may end up referring to the same event or program location: all of them are run in an unspecified sequence when the probe point is hit. For tapset builders, there is also a probe aliasing mechanism discussed in section 4.1

An auxiliary function is a subroutine for probe handlers and other functions. In order to conserve memory, there may be a limit imposed on the number of outstanding nested or recursive calls. The translator provides a number of built-in functions, which are implicitly declared and listed in section 3.5.

A global variable declaration lists variables that are shared by all probe handlers and auxiliary functions. (If a variable is not declared global, it is assumed to be local to the function or probe that references it.)

A script may make references to an identifier defined elsewhere in library of script tapsets. Such a cross-reference causes the entire tapset file providing the definition to be merged into the elaborated script, as if it was simply concatenated. See section 4 for more information about tapsets.

Fatal errors that occur during script execution cause a winddown of activity associated with the systemtap script, and an early abort. Running out of memory, dividing by zero, exceeding an operation count limit, calling too many nested functions, are just a few types of fatal errors.

3.1 Probe points

A probe definition gives probe points in a comma-separated list, and an associated action in the form of a statement block. A trigger of any of the probe points will run the block. Each probe point specification has a "dotted-functor" syntax such as `kernel.function("foo").return`. The core systemtap translator recognizes a family of these patterns, and tapsets may define new ones. The basic idea of these patterns is to provide a variety of user-friendly ways to refer to program spots of interest, which the translator can map to a kprobe on a particular PC value or an event setup API.

The first group of probe point patterns relates to program points in the kernel and kernel modules. The first element, `kernel` or `module("foo")`, identifies the probe's target software (a kernel, or a kernel module named "foo.ko"), and is used to find the symbolic debug information to resolve the rest of the pattern.

An important fact associated with probe points defined on statically known program elements is that the translator can find debugging information relevant to each spot. It can then expose local variables within the scopes of these functions to the script.

3.1.1 Functions

To identify a function, the `function("fn")` element does so by name. If function is inlineable, all points of inlining are included in the set. The function name may be suffixed by `@filename` or even `@filename:lineno` to identify a source-level scope within which the identifiers should be searched. The function name may include wildcard characters `*` and `?`, to refer to all suitable matching names. These may expand to a huge list of matches, and therefore must be used with discretion.

Alternately, with a numeric parameter, `function(nnnn)` may identify an absolute PC address for kernel, or a load-address-relative address for kernel module. The function identified this way is the one containing the address in the symbol table, without regard for inlining. Such an address is of course architecture- and build-specific, so must be used with care.

Next, the optional element `callees` replaces the set of matched functions with the set of all functions statically known to be callable from the first set. Finally, the optional element `return` may be added to refer to

the moment of each function's return rather than the default `entry`.

3.1.2 Statements

Instead of specifying a function's entry or exit, one can refer to a particular statement within a function at which to place a probe point. This makes it possible to look deeper into the control flow, to examine local variables. The statement is identified by the probe point element `statement("foo")`, where "foo" is parsed much like a function probe. It starts with a function name, and ends with an optional source file and line number. This would refer to the first instruction of the statement at or after the given line.

Because specifying an absolute line number is clumsy and fragile with respect to changes in kernel versions, a few other ways are available. First, instead of an absolute line number, one may add a `relative(NN)` element to indicate an offset relative to the first line of the function. Second, one may add the name of a label after the name of the function containing it: `label("need_resched")`. In this case, the probe point is placed at the first statement just after the label.

An absolute address syntax is also available for statement probe points, just like for functions: `statement(nnnn)` is interpreted as referring to the the kernel (or module) instruction that spans the given absolute (or relative) address.

3.1.3 Events

Probe points may be defined on abstract events, which are not associated with a particular point in the target program. Therefore, the translator cannot expose much symbolic information about the context of the probe hit to the script.

Use the special element `begin` to trigger a probe handler early during systemtap initialization, before normal probes are enabled. Similarly, `end` triggers a probe during late shutdown, after all normal probes have been disabled.

3.1.4 Examples

Here are some examples:

`kernel.function("sys_read").return`
: a return probe on the named function.

`module("ext3").function("*@fs/ext3/inode.c")`
: every function in the named source file, a part of ext3fs

`kernel.function("kmalloc").callees`
: every function known statically to be callable from kmalloc

`module("usb-storage").statement(0x0233)`
: the given address, which must be at an instruction boundary

`kernel.function(0xffffffff802202dc).return`
: a return probe on whichever function that contains the given address

3.2 Language Elements

Function and probe handler bodies are defined using standard statement/expression syntax.

Identifiers Systemtap identifiers have the same syntax as C identifiers, except that `$` is also a legal character. Identifiers are used to name variables and functions. Identifiers that begin with `$` are interpreted as references to variables in the target software, rather than to systemtap script variables.

Types The language includes a small number of data types, byt no type declarations: a variable's type is inferred from its use. To support this, the translator enforces consistent typing of function arguments and return values, array indexes and values. Similarly, there are no implicit type conversions between strings and numbers.

- Numbers are 64-bit signed integers. Literals can be expressed in decimal, octal, or hexadecimal, using C notation. Type suffixes (e.g., `L` or `U`) are not used.
- Strings. Literals are written as in C. Overall lengths may be limited by the runtime system.
- Statistics. These are special objects that compute aggregations (statistical averages, minima, histograms, etc.) over numbers.
- Associative arrays, as in awk. A given array may be indexed by any consistent combination of strings and numbers, and may contain strings, numbers, or statistical objects.

Semicolons Semicolons are used to optionally separate statements. The grammar does not require formal statement separators/terminators, but their presence helps clarify parser error messages.

Comments A comment may take any of the following forms:

- `/*` ... `*/`, as in C.
- `//` ... end-of-line, as in gcc and C++
- `#` ... end-of-line, as in awk and shells

White Space As in C, spaces, tabs, returns, newlines, and comments are treated as white space.

3.3 Statements

Systemtap the following types of statements, which have the same syntax and semantics as in C:

- `break`
- `continue`
- `for ( <expression> ; <condition> ; <expression> ) <stmt>`
- `if ( <condition> ) <stmt> [ else <stmt> ]`
- `while ( <condition> ) <stmt>`
- `<expression>` (Note that this includes assignment statements.)
- `;` (null statement)
- `{ <stmt> ... }` (statement block with zero or more statements)

Systemtap also supports the following types of awk statements:

`foreach ( <names> in <array_name> ) <stmt>`
As in awk, iterates through all the keys in an associative array, with no particular order guaranteed. Each key in turn is assigned to `<name>`, and the specified statement (block) is executed. A multidimensional array requires a matching number of index arguments, enclosed in square brackets: `[a, b, c]`.

`return <expression>`
Returns from a function. Unlike in C, a return value is required. This is because the void type does not exist.

`next`
Returns from a probe handler.

`delete <expression>`
Deletes the contents of an entire array, or only an element at a given index.

```
delete noise
delete smell ["dog", 0]
```

Systemtap does not support `goto` statements or labels, `switch` statements.

3.4 Expressions

In general, systemtap expressions follow the same syntax and semantics as C expressions. The following operators have the same meaning as in C:

- arithmetic/bit operators: `* / % + - >> << & ^ |`
- comparison operators: `< > <= >= == != && ||`
- assignment operators: `= *= /= %= += -= >>= <<= &= ^= |=`
- unary operators: `- ! ~ ++ --`
- the ternary operator: `<condition> ? <expr> : <expr>`
- parentheses `( )` for grouping
- function calls. See section 3.5

Systemtap also supports the following operators:

- statistics accumulation: `<<<`
  Adds a given numeric value to a statistics object. Example:

```
global avg(s)
probe kernel.syscall("read") {
  process->s <<< $size
}
probe end {
  trace (s)
}
```

This syntax is being discussed for revision.

- string concatenation: `.` (period) as in `perl`.

```
path = dirname . "/" . basename . "." . suffix
bad_names .= ", " . this_name
```

- string comparison, assignment: These overload the usual `=`, `<`, etc. operators.
- `<key> in <array>` or `[<key1>, <key2>, ...] in <array>`. Example:

```
delete bad_days
bad_days["Mon"] = BD_LOUSY
activity["Wed","shopping"] = 1
if ("Mon" in bad_days) report("no surprise")
if (["Tue"] in bad_days) report("internal error")
if (["Wed","shopping"] in activity) report("popcorn")
```

- associative-array references: `<array>[<expr>]` or `<array>[<expr>,<expr>]`
- `$<var>-><field>`. Here `<var>` is a reference to a struct-pointer type variable in the target.

Systemtap does not support the following operators:

- `<struct> . <field>`
- `* <pointer>`
- `<pointer> -> <field>` (But see `<macro> -> <field>`.)
- `& <lvalue>`
- `sizeof`
- type casts – e.g., `(long) val`
- `,` (comma operator)

3.5 Auxiliary functions

An auxiliary function in systemtap has essentially the same syntax and semantics as in awk. Specifically, an auxiliary function definition consists of the keyword `function`, a formal argument list, followed by a brace-enclosed statement block.

Systemtap deduces the types of the function and its arguments from the expressions that refer to the the function.

4 Tapsets

Systemtap "tapsets" are abstraction constructs for use by scripts, written in script language or C, and stored in a library for use during the elaboration phase. There are several types of tapsets, depending on which kinds of extension facilities they use.

4.1 Script tapsets

The simplest kind of tapset is one that uses the ordinary script language to define new probes, auxiliary functions, global variables, for invocation by an end-user script or another tapset. Recall that a script that makes otherwise undefined reference to an identifier (function or global variable) that is defined by another script in a library directory causes that script to be included in the elaborated program.

One can use this mechanism to define a useful auxiliary function centrally, such as routines to compute useful mathematical or logical functions:

```
function log2(val) {
  for (i=0; val>0; val /= 2)
    i++;
  return i;
}
```

Similarly, a clever tapset author can provide "automagic" global variables, as if they were built-in:

```
global pid2path # always contains a map of active executables

probe kernel.function("sys_execve") {
  pid2path [$pid] = $name
}
probe kernel.function("sys_exit") {
  delete pid2path [$pid]
}
```

It can be more elaborate:

```
global tgid_history # always contains the last few tgids scheduled

global _histsize
probe begin {
  _histsize = 10
}
probe kernel.function("context_switch") {
  # rotate array
  for (i=_histsize-1; i>1; i--)
    tgid_history [i] = tgid_history [i-1];
  tgid_history [0] = $prev->tgid;
}
```

In addition, a script tapset can define a *probe alias*. This is a way of synthesizing a higher level probe out of a lower level one. This consists of renaming a probe point, and may include some script statements. These statements are all executed *before* the others that are within the user's probe definition (which referenced the alias), as if they were simply transcribed there. This way, they can prepare some useful local variables, or even conditionally reject a probe hit using the `next` statement.

The following tapset defines aliases for system calls, so that a systemtap user does not have to remember which kernel functions implement the abstract POSIX system calls.

```
probe kernel.syscall.read =
kernel.function("sys_read") { }

probe kernel.syscall.fork =
kernel.function("sys_fork") { }
```

The following script tapset defines a new “event”, and supplies some variables for use by its handlers.

```
probe kernel.resource.oom.nonroot =
kernel.statement("do_page_fault").label("out_of_memory") {
   if ($tsk->uid == 0) next;

   victim_tgid = $tsk->tgid;
   victim_pid = $tsk->pid;
   victim_uid = $tsk->uid;
   victim_fault_addr = $address
}
```

A script that uses this probe alias may look like this:

```
probe kernel.resource.oom.nonroot {
  trace ("OOM for pid " . string (victim_pid))
}
```

4.2 C tapsets

TBD

5 Safety

Systemtap is designed to be safe to use on production systems. An implication is that it should be extremely difficult if not impossible to disable or crash a system through use or misuse of Systemtap. Problems like infinite loops, division by zero, and illegal memory references should lead to a graceful failure of a Systemtap script without otherwise disrupting the monitored system. At the same time, we’d like for Systemtap extensions to compile to machine code, to leverage existing infrastructure such as compilers and ”insmod” and to approach native performance.

Our basic approach to safety is to design a safe scripting language, with some safety properties supported by runtime checks. Table 1 provides some details of our basic approach. Systemtap compiles the script file into native code and links it with the systemtap runtime library to create a loadable kernel module. Version checks and symbol name checks are applied by insmod [I ASSUME WE GET THE SAME SAFETY HERE AS /sbin/insmod]. The elaborator generates instrumentation code that gracefully terminates loops and recursion at runtime if they run beyond a configurable threshold. We avoid priviledged and illegal kernel instructions by excluding constructs in the script language for inlined assembler, and by using compiler options commonly used for the kernel.

	language design	translator	insmod checks	runtime checks	memory portal	static validator
infinite loops		x		o		o
recursion		x		o		o
division by zero		x		o		o
resource constraints		x		x		
locking constraints		x		x		
array bounds errors	x	x		x		o
invalid pointers		o		o		o
heap memory bugs	x					o
illegal instructions	x				o	
privileged instructions	x				o	
memory r/w restrictions	x			x	o	o
memory execute restrictions	x			x	o	o
version alignment		o	x			
end-to-end safety					x	x
safety policy specification facility					x	

Table 1: Systemtap safety mechanisms. An “x” indicates that an aspect of the implementation (columns) is used to implement a particular safety feature (rows). An “o” indicates optional functionality.

Systemtap incorporates a number of design features that enhance safety. Explicit dynamic memory allocation by scripts is not allowed, and dynamic memory allocation by the runtime is discouraged. Systemtap can frequently use explicitly synthesized frames in static memory for local variables, avoiding usage of kernel stack. Language and runtime systems ensure strictly terminating, nonblocking body code in probes.

Systemtap safety also requires controling access to kernel memory. Kernel code cannot be invoked directly from a Systemtap script. The Systemtap runtime can use kernel subroutines, and these references are assumed to be safe. Systemtap provides special language features for referring to external data. [PROBLEM: POINTER ARGUMENTS TO PROCEDURES] When it sees such references, it constrains it based on a policy specified when the script is run. By default, writes to arbitrary kernel memory are prohibited, and external reads of any kernel memory are allowed except for the range of the virtual address space used for memory-mapped devices.

To implement these restrictions, external writes cause the elaborator to generate an error, and checking code is inserted to enforce range restrictions on external read references. Additionally, a modified trap handler is used to safely handle invalid memory references anywhere in the kernel address space. Systemtap also supports a ”guru” mode where these code and data reference constraints are removed. This allows us to tradeoff safety features to support the needs of kernel debugging tasks.

5.1 Safety Enhancements

Our use of established tools and minimizing the amount of new code added to the kernel significantly enhances the safety of Systemtap.

We are considering a number of subsystems that extend the safety and flexibility of Systemtap to match

and exceed that of other systems based on interpreters. A memory and code ”portal” would direct external memory references from Systemtap scripts, both to code and data, through a special-purpose interpeter or ”portal.” The goal of the portal is to shift safety policy selection to the script user rather than the script author, and to support definition of safety policies by script users, giving them explicit and finer-grained controls. We anticipate a model where people commonly use scripts written by others.

Trivial policies would be provided with the system to support ”guru” (no restrictions) and default modes (restricted external write and code access). Other simple policies would expand access for a script incrementally. For example, you might allow external calls to an explicit list of kernel subroutines, or writes to an explicit list of kernel data structures or range of kernel memory addresses. Such policies would expand script capabilities with better safety than guru mode. Eventually, the policy subsystem might be extended to support security goals, such as restricting memory access control based on UID.

A optional static analyzer provides redundant checking and protection against various system bugs, installation problems and misuse. The static analyzer examines a disassembled kernel module and confirms that it satisfies certain safety properties. Simple checks include disallowing priviledged instructions and instructions that are illegal in kernel mode. More elaborate checks confirm that loop counters, memory portals and other safety features are used, given minimal cooperation from the language implementation to make the machine code checkable. The analyzer rejects uncheckable code.

5.2 Comparision to Other Systems

Solaris DTrace includes a number of unusual features intended to enhance the safety and security of the system. These features include:

very restricted scripting language
: The D language does not support procedure declarations or a general purpose looping construct. This avoids a number of safety issues in scripts including infinite loops and infinite recursion.

interpreted language
: The D scripts compile to a RISC abstract machine language that executes in an interpreter embedded in the Solaris kernel. Because D scripts are interpreted rather than executed directly, it is impossible for them to include illegal or priviledged instructions or to invoke code outside of the DTrace execution environment. The interpreter can also catch invalid pointer dereferences, division by zero, and other run-time errors.

Features such as these enhance the perceived safety of DTrace.

Systemtap will support kernel debugging features that DTrace does not, including ability to write arbitrary locations in kernel memory and ability to invoke arbitrary kernel subroutines.

Since the language infrastructure used by Systemtap is common to all C programs, it may be better tested and more robust than the special-purpose infrastructure used by DTrace.

The embedding of an interpreter in the Solaris kernel represents significant additional kernel functionality. This introduces an increased risk of kernel bugs that could lead to security or reliability issues.

DProbes and DTrace have many safety features in common. Both use an interpreted language. Both use a modified kernel trap-handler to capture illegal memory references. DProbes exposes the KProbes layer in such a way that it is not crashproof, as it does allow invalid instrumentation requests.

Would a DProbes expert like to finish this part?

5.3 Security

It is important that Systemtap can be used without significantly impacting the overall security of the system. We assume for initial releases that root priviledges are required for Systemtap use.

Given that Systemtap is only available to priviledged users, our initial security concerns are that the system be crash-proof by design, and that its implementation is of sufficient quality and simplicity to protect users from unintentional lapses. A specific concern is security of communication layer; that the kernel-to-user transport is secured from unpriviledged users.

Future versions of Systemtap may provide features that support secure use of Systemtap by unpriviledged users. Specific features that might be required include:

protection of kernel memory based on user credentials

protection of kernel-to-user transport based on user credentials

the compilation system might recognize a restricted subset of the Systemtap language that is permissible for non-priviledged users

A virtual-machine based security scheme might provide a simpler and more general solution to secure Systemtap use by unpriviledged users.

6 Lower layer issues

6.1 Kernel-to-user transport

Data collected from systemtap in the kernel must somehow be transmitted to userspace. This transport must have high performance and minimal performance impact on the monitored system.

One candidate is `relayfs`. Relayfs provides an efficient way to move large blocks of data from the kernel to userspace. The data is sent in per-cpu beffers which a userspace program can save or display. Drawbacks are that the data arrives in blocks and is separated into per-cpu blocks, possibly requiring a post-processing step that stitches the data into an integrated steam.

Relayfs is included in some recent -mm kernels. It can be built as a loadable module and is currently checked into CVS under `src/runtime/relayfs`.

The other candidate is `netlink`. Netlink is included in the kernel. It allows a simple stream of data to be sent using the familiar socket APIs. It is unlikely to be as fast as relayfs.

Relayfs typically makes use of netlink as a control channel. With some simple extensions, the runtime can use netlink as the main transport too. So we can currently select in the runtime between relayfs and netlink, allowing us to support streams of data or blocks. And allowing us to perform direct comparisons of efficiency.

6.2 Output

Depending on the primitives used in the systemtap script, output may flow gradually via logging streams (`printk`, `netlink`, etc.), or in large batches (`relayfs` files). In some cases, systemtap would infer the relationship between arrays, indexes, and automatically format related results in a naturally combined way. For example, if systemtap notices that three separate arrays are always indexed by the same variable, in the output it can combine the three arrays into a four-column listing, sharing the index rows.

Other than a simple textual form, systemtap should also be able to emit the overall data in a structured computer-parsable form such as XML, or into other forms easily loaded by graphics generator programs. This step would happen in the post-processing phase by a userspace utility.

References

[1] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Levinthal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Technical Conference*, pages 15–28, June 2004.

[2] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *FREENIX*, 2001.