put systemtap band-aids on security wounds

Frank Ch. Eigler

Red Hat
fche@redhat.com

FOSDEM 2016-01-30
Some security bugs can be patched with systemtap. Instead of changing code, change data on the fly.
What’s the matter? New CVE giving you a bad day?
Security bugs:
- happen
- can rarely ignore
- probably require patches
- source patches need healthy upstream
- patched binaries need healthy distributor
- will require restarting fixed services
the common solution: patch code

- patch source (roll your own?)
- rebuild binary (or live patch?)
- distribute binary
- restart machine or services

What if one of these is too difficult or too slow?
an uncommon solution: patch data

- code not run is not a problem

- four cases:

<table>
<thead>
<tr>
<th>code</th>
<th>data friendly</th>
<th>data hostile</th>
</tr>
</thead>
<tbody>
<tr>
<td>good</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>vulnerable</td>
<td>!</td>
<td>⚠</td>
</tr>
</tbody>
</table>

- we want to avoid the ⚠️
- a code patch moves system ▲
- a data patch moves system ◀
- the ⚠️ state may be safe
How would you patch data?
what is systemtap

- for tracing
- for profiling
- for understanding
- for debugging
- the whole system
- the linux kernel
- C/C++/Go/Java/Python userspace
- at source and binary level
more about systemtap

- is kind of like dtrace
- is kind of like scripted gdb
- is kind of like “if this then that”
- is programmable, not hard-coded
- is powerful, performant
- is transparent, safe
- is *normally* read-only
domain-specific programming language

concept: when event happens, do something, then resume

```python
probe POINT { HANDLER }
```

many kinds of probe points

arbitrary probe handlers, run atomically

expressive control flow

rich data management

compact for quick “one-liners”

light on punctuation, heavy on automation
probe points: when to act

- probe points name events: when to act

```plaintext
begin, end
timer.profile
kernel.function("sys_open").call
process("/lib*/libc.so.6").statement("*@malloc.c:75")
kernel.trace("sched:sched_wakeup")
syscall.read*
netfilter.ipv4.local_in
process("/usr/bin/qemu*").mark("cpu_in")
perf.hw.bus_cycles.sample(123456)
```

- ... and more!
- ... at the same time
- see [man stapprobes] for details
probe handlers: how to act

- probe handlers specify code to run: how to act upon the data

```c
hit_count ++
printf("%s %d %s\n", execname(), pid(), $$vars)
for (i = 0; i < $buf; i++) total += $foo[i]
if (randint(10) < 1) print_backtrace()
if ($var > 0) $var = 0
```

- ... and more!
- does *not* replace original code at probe point
- see [man stap] or language reference guide for details
automation facilities

- analysis of DWARF debuginfo for probe target, \$variables
- pretty-printing
- strict and implicit typing, type inference
- locking of shared variables
- variable initialization
- functions, macros, look-up tables
- strict time & space limit enforcement
- auto-\#including tapset: abstract away architectures / versions
- ... and more!
implementation

- compilation
  - analyze, type-infer, check
  - compile to checked C
  - compile to kernel module

- execution
  - load as kernel module (or other backend)
  - attach to event sources, system-wide, online
  - run, relay outputs
  - unload & clean up
an example

```bash
# cat strace-nonroot.stp
probe nd_syscall.*
{
    if (uid() != 0)
        printf("%s %d %s (%s)\n", execname(), tid(), name, argstr)
}
# stap strace-nonroot.stp -c true
roxterm 421 read (5, 0x7fffc3f09a6c0, 16)
roxterm 421 recvmsg (6, 0x7fffc3f09a510, 0x0)
roxterm 421 poll (0x43049f0, 15, 10)
Timer 3989 futex (0x2aaef4683690, FUTEX_WAKE_PRIVATE, 1)
Timer 3989 write (80, "\372", 1)
[...]  
See http://sourceware.org/systemtap/examples/ for 145 more.
any modern reasonably recent kernel: 2.6.18 ... present
- kconfig including kprobes, tracepoints, uprobes, etc.
- the newer compiler the better, some distros lag
- the more debuginfo the better, some distros lose
- the newer systemtap the better, some distros lag
- remote compilation: stap-server
- remote or deferred execution: staprun only
- unprivileged execution: stapusr, dyninst
OK, how would you patch hostile data with systemtap?
strategy

- study vulnerable piece of code
- analyze conditions for hostile data
- draft algorithm to make the hostile data safe, or to reject it
- express algorithm in systemtap script
- run it under "guru" mode: `# stap -g SCRIPT.stp`
- with root approval, permits $var writing, embedded-C
some disclaimers

- this is not always easy: analysis, programming
- this is not always applicable
- beware a cunning enemy: the optimizing compiler
- may require trial & error
- this is not always convenient: logistics
- this is not without risk
- this is a temporary solution
optimizing compilers - the enemy

- deployed object code fully optimized
- scrambles code (inlining, reordering)
- scrambles data (live regions, elision, folding)
- limits systemtap probe placement & data availability
- `stap -L PROBEPPOINT` to list probe points & variables
- *good-quality* debuginfo needed
- `gcc -g -fvar-tracking-assignments` is avant-garde
- `llvm` not in the same league
- friends don’t let friends degrade, strip, or lose debuginfo!
## Applicability Analysis

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Patch Code</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code available</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vulnerability analyzed</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Localized bug</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Simple control flow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bug depends on local data</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data accessible</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bug conditions unambiguous</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Few of the above</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
How to mitigate a simple buffer-overflow?

```c
void fn (char *buf) {
    char buf2[20];
    strcpy (buf2, "hello");
    strcat (buf2, buf);
    printf ("%s", buf2);
}
```
Find the bug & trigger conditions.

```c
1  void fn (char *buf) {
2      char buf2[20];
3      strcpy (buf2, "hello");
4      strcat (buf2, buf); /* <- bug here, if buf long */
5      printf ("%s\n", buf2);
6   }

# ./gedank1
hello everything is awesome when you’re on the world
[2] 22525 segmentation fault (core dumped) ./gedank1
```
Put a NUL at \texttt{buf[14]}, assuming it is writeable.

```c
probe process\_statement("*@*:4") {
    buf = user\_string($buf)
    if (strlen(buf) >= 14)
        $buf[14] = 0
}
```

# stap -g gedank1.stp -c ./gedank1

```
hello everything i
```

```c
global saved
probe process.statement("*@*:4") {
    buf = user_string($buf)
    if (strlen(buf) >= 14)
        saved[tid()] = $buf[14]
        $buf[14] = 0
}
}
probe process.statement("*@*:5") {
    if (tid() in saved) {
        $buf[14] = saved[tid()]
        delete saved[tid()]
    }
}
```

# stap -g gedank2.stp -c ./gedank2

hello everything i
everything is awesome when you’re on the world
What if `buf` was read-only? Redirect it to another variable whose content we can influence.

```plaintext
probe process.statement("*@*:4") {
    buf = user_string($buf)
    if (strlen(buf) >= 14)
        $buf = & @var("another")
}
```

# stap -g gedank3.stp -c ./gedank3
hello
How to mitigate a poor check?

```c
int foo (char *buf) {
    char buf2[20];
    int buflen = strlen(buf);
    if (buflen > 20)
        goto err;
    strcpy (buf2, "hello");
    strcat (buf2, buf);
    printf ("%s\n", buf2);
    return 0;
err:
    return -1;
}
```
Find the bug & trigger conditions.

```c
01 int foo (char *buf) {
02     char buf2[20];
03     int buflen = strlen(buf);
04     if (buflen > 20) /* <- bug here, wrong number */
05         goto err;
06     strcpy (buf2, "hello");
07     strcat (buf2, buf);
08     printf ("%s\n", buf2);
09     return 0;
10 err:
11     return -1;
12 }
```

Quiet buffer overflow!

```bash
# ./gedank4 67890123456789012345
hello67890123456789012345
rc=0
```
Increment buflen to account for string-processing

/* between assignment and comparison * /
probe process.statement("*@*:4") {
    $buflen = $buflen + 6
}

# stap -g gedank4.stp -c "./gedank4 67890123456789012345"
rc=-1
# stap -g gedank4.stp -c "./gedank4 67890123456789"
hello67890123456789
rc=0
choose a mitigation

- option 1: redirect control flow to error-handling path
- option 2: correct data, permit normal control flow
  both options may be available

<table>
<thead>
<tr>
<th>attributes</th>
<th>preferred mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple to correct data</td>
<td>✓</td>
</tr>
<tr>
<td>hostile data likely</td>
<td>✓</td>
</tr>
<tr>
<td>correction probe points/vars available</td>
<td>✓</td>
</tr>
<tr>
<td>error-handling path nearby</td>
<td>✓</td>
</tr>
<tr>
<td>high-quality debuginfo</td>
<td>✓</td>
</tr>
</tbody>
</table>
QEMU floppy-drive emulation bug: buffer overflow via range-unchecked value: fdctrl->data_pos in function fdctrl_handle_drive_specification_command.

Patch excerpt:

```c
- if (fdctrl->fifo[fdctrl->data_pos - 1] & 0x80) {
+ uint32_t pos;
+ pos = fdctrl->data_pos - 1;
+ pos %= FD_SECTOR_LEN;
+ if (fdctrl->fifo[pos] & 0x80) {
```
Impose bounds on suspect variable. Restore it afterwards.

global saved_data_pos
probe process("/usr/bin/qemu-system-*")
  .function("fdctrl_*spec*_command").call
{
  saved_data_pos[tid()] = $fdctrl->data_pos;
  $fdctrl->data_pos = $fdctrl->data_pos % 512
}
probe process("/usr/bin/qemu-system-*")
  .function("fdctrl_*spec*_command").return
{
  $fdctrl->data_pos = saved_data_pos[tid()]
  delete saved_data_pos[tid()]
}
Realize all FDC access is suspect. Reroute all writes to a reserved NOP register.

```plaintext
probe process("/usr/bin/qemu-system-*")
    .function("fdctrl_write")
{
    $reg = (($reg & ~7) | 6) # redirect to 0x__6
}
```
So far, we mitigated 100% of those CVEs we tried, fingers crossed.
other mitigation-related options

- trace suspicious data
  ```
  printf("check buffer \%.\*M", $size, $buf)
  ```

- report attack to system logs

- leave “honeypot” even after code patched
  ```
  system(sprintf("/bin/logger pid %d attacked", pid()))
  ```

- kill victim process
  ```
  raise(9) /* SIGKILL */
  ```
Anything else?
some other uses

non-guru mode
- gather algorithm internal statistics
- spying on programs or users
- per-function or per-statement tracing, profiling

guru mode
- secure clear memory after crypto, or free() for ksm
- patch in hardware support via USB vendor tables
- fix non-security bugs
- tune adaptive algorithms
for more information

- systemtap@sourceware.org
- #systemtap on irc.freenode.net
- https://sourceware.org/systemtap/
- https://securityblog.redhat.com/2015/06/03/emergency-security-band-aids-with-systemtap/