put systemtap band-aids on security wounds

Frank Ch. Eigler

Red Hat
fche@redhat.com

FOSDEM 2016-01-30

abstract

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?

common knowledge

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:

		data	
		friendly	hostile
code	good	✓	✓
	vulnerable	⚠	💣

- we want to avoid the 💣
- a code patch moves system $\Uparrow$
- a data patch moves system $\Leftarrow$
- 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

theory of operation

- domain-specific programming language
- concept: when event happens, do something, then resume

```
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

```
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

```
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

```
# 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, 0x7ffc3f09a6c0, 16)
roxterm 421 recvmsg (6, 0x7ffc3f09a510, 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.

deployment

- 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 PROBEPOINT` 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

attributes	patch	
	code	data
source code available		✓
vulnerability analyzed		✓
localized bug		✓
simple control flow		✓
bug depends on local data		✓
data accessible		✓
bug conditions unambiguous		✓
few of the above	✓	

mitigation exercise one

How to mitigate a simple buffer-overflow?

```
void fn (char *buf) {
    char buf2[20];
    strcpy (buf2, "hello");
    strcat (buf2, buf);
    printf ("%s", buf2);
}
```

mitigation exercise one - analysis

Find the bug & trigger conditions.

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

```
# ./gedank1
hello everything is awesome when you're on the world
[2]    22525 segmentation fault (core dumped)  ./gedank1
```

mitigation exercise one - solution one

Put a NUL at `buf[14]`, assuming it is writeable.

```
probe process.statement("*@*:4") {
  buf = user_string($buf)
  if (strlen(buf) >= 14)
     $buf[14] = 0
}
```

```
# stap -g gedank1.stp -c ./gedank1
hello everything i
```

mitigation exercise one - solution two

What about restoring buf[14] afterwards? Store it in a global.

```
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
```

mitigation exercise one - solution three

What if `buf` was read-only? Redirect it to another variable whose content we can influence.

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

```
# stap -g gedank3.stp -c ./gedank3
hello
```

mitigation exercise two

How to mitigate a poor check?

```
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;
}
```

mitigation exercise two - analysis

Find the bug & trigger conditions.

```
int foo (char *buf) {
  char buf2[20];
  int buflen = strlen(buf);
  if (buflen > 20) /* <- bug here, wrong number */
    goto err;
  strcpy (buf2, "hello");
  strcat (buf2, buf);
  printf ("%s\n", buf2);
  return 0;
err:
  return -1;
}
```

Quiet buffer overflow!

```
# ./gedank4 678901234567890123456
hello678901234567890123456
rc=0
```

mitigation exercise two - solution

Increment `buflen` to account for string-processing

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

```
# stap -g gedank4.stp -c "./gedank4 678901234567890l2345"
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

attributes	preferred mitigation	
	induce error	correct data
simple to correct data		✓
hostile data likely	✓	
correction probe points/vars available		✓
error-handling path nearby	✓	
high-quality debuginfo		✓

example: VENOM CVE-2015-3456 - bug/patch

QEMU floppy-drive emulation bug: buffer overflow via range-unchecked value: `fdctrl->data_pos` in function `fdctrl_handle_drive_specification_command`.

Patch excerpt:

```
-    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) {
```

example: VENOM CVE-2015-3456 - correct data

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()]
}
```

example: VENOM CVE-2015-3456 - induce error

Realize all FDC access is suspect. Reroute all writes to a reserved NOP register.

```
probe process("/usr/bin/qemu-system-*")
      .function("fdctrl_write")
{
    $reg = (($reg & ~7) | 6) # redirect to 0x__6
}
```

track record

CVE	module	description	mitigation
CVE-2008-0600	kernel	vmsplice input validation	error
CVE-2012-0056	kernel	/proc mem_write perm checking	error
CVE-2013-2094	kernel	perf_swevent_enabled overflow	correct
CVE-2014-7169	bash	SHELLSHOCK remote execution	error
CVE-2015-0235	glibc	GHOST nss buffer overflow	correct
CVE-2015-3456	qemu	VENOM fdc overflow	either
CVE-2016-0728	kernel	key refcount overflow	correct

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/