SYNACKTIV

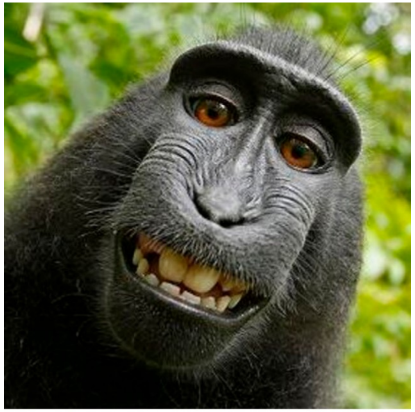
Real hackers don't leave DTrace

Sthack 2022

20/05/2022

Whoami

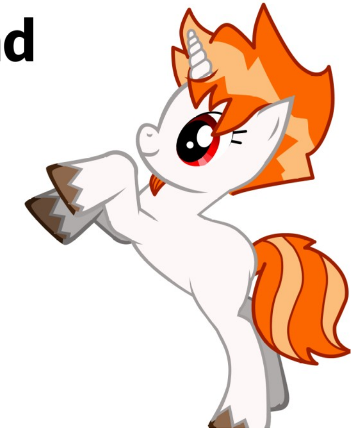
- **Eloi Benoist-Vanderbeken**
  - @elvanderb on twitter
- **Working for Synacktiv**
  - Offensive security company
  - 100+ ninjas
  - 4 departments
    - Pentest
    - Reverse engineering
    - Development
    - IR
  - Sthack sponsor!
- **Reverse engineering technical lead**
  - 30 reversers
  - Focus on low level dev, reverse, vulnerability research/exploitation
  - If there is software in it, we can own it :)
  - We are hiring!



DTrace

- **Created by Sun Microsystems for Solaris in 2003**
  - macOS/XNU in 2007
  - Linux in 2008
  - FreeBSD and NetBSD in 2009/2010
  - Windows in 2019
- **https://dtrace.org**

- **Designed to debug problems…**
- **… on production systems…**
  - No special configuration needed
- **… dynamically…**
  - No need to recompile anything
- **… in user and kernel land**
  - Very powerful!



DTrace

“allows you to ask arbitrary questions about what the system is doing, and get answers”

Bryan Cantrill, DTrace coinventor

DTrace

“DTrace is a magician that conjures up rainbows, ponies and unicorns — and does it all entirely safely and in production!”

Original fill-me-in text of the DTrace about page

What it does

- **Execute code**
  - Special language: D
  - C-like
  - No loops
  - No branches
    - But conditional expressions
  - Can inspect the program state
  - Can execute actions
  - Can use variables and aggregations

- **Enable probes**
  - “hooks”
  - Provided by… providers
    - Lots of them
  - Can execute D code when hit
  - Execution can be conditioned by a predicate

Providers

- **PID**
  - Target a process
- **FBT**
  - Dynamic hooks in the kernel
- **Profile**
  - Provide periodic samples
- **Syscall**
  - Who execute syscalls
- **IO**
  - Disk input and output
- **Sched**
  - Scheduler information
- **mach_trap**
  - Same than syscall but for mach traps
- **objc_runtime**
  - Interaction with Objective C
- **Magmalloc**
  - Internals of macOS allocator
- **Python**
- **…**

How it is used

- **Mostly to debug performance problems**
  - Production systems problems
  - Aggregations make that really easy
- **But it can be used for a lot of other things**
  - Reversing
  - Exploit
  - Threat hunting
  - etc.

Warning

The rest of the presentation is focused on macOS

Example


```
user@mac:~$ sudo dtrace  -n 'proc:::exec {
>       printf("%s execute %s\n", execname, (string)args[0]);
>       ustack(5);
> }'

dtrace: description 'proc:::exec ' matched 1 probe

CPU     ID                    FUNCTION:NAME
  2   1496                posix_spawn:exec launchd execute /usr/libexec/xpcproxy

              libsystem_kernel.dylib`__posix_spawn+0xa
              libsystem_c.dylib`posix_spawnp+0x1b1
              launchd`0x0000000106983831+0xacf
              launchd`0x000000010698c241+0x41
              libdispatch.dylib`_dispatch_client_callout+0x8
```

Example

```
user@mac:~$ sudo dtrace  -n 'pid1:libsystem_malloc:malloc:entry {
>       printf("launchd calls malloc(%d)", arg0);
>       ustack(3);
> }'

dtrace: description 'pid1:libsystem_malloc:malloc:entry ' matched 1 probe

CPU     ID                    FUNCTION:NAME
  1 947427                       malloc:entry launchd calls malloc(184)
              libsystem_malloc.dylib`malloc
              libsystem_kernel.dylib`posix_spawnattr_init+0x13
              launchd`0x000000010697b219+0x30
```

Example


```
user@mac:~$ sudo dtrace  -n 'pid1:libsystem_malloc:malloc:entry {
>       self->malloc_size = arg0;
>   }' \
> -n 'pid1:libsystem_malloc:malloc:return
>       / self->malloc_size > 1024 / {
>       printf("malloc(%d) returned 0x%X\n", self->malloc_size, arg1);
>   }'

[…]

CPU     ID                    FUNCTION:NAME
  2 947426                      malloc:return malloc(2088) returned 0x7FFF811C07D0

  1 947426                      malloc:return malloc(2877) returned 0x7000099B40AC
```

Example

user@mac:~$ sudo dtrace -n 'pid1:libsystem_malloc:malloc:entry { @sizes = quantize(arg0); }'

```
        value  ------------- Distribution ------------- count
            1 |                                         0
            2 |                                         117
            4 |                                         184
            8 |                                         187
           16 |@@@                                      1391
           32 |@@@@@@@@@@@@@@@@@                        7100
           64 |@@@@@@@@@@@@@@@@                         6922
          128 |@@                                       961
          256 |                                         0
          512 |                                         0
         1024 |                                         10
         2048 |                                         12
         4096 |                                         0
```

Example


```
user@mac:~$ sudo dtrace -n 'profile:::profile-1001
>     /execname == "Xcode"/ {
>     @[ustack(5)] = count();
> }'

[...]
              libsystem_kernel.dylib`mach_absolute_time+0x1c
              QuartzCore`+[CATransaction(CATransactionPrivate) generateSeed]+0x3c
              AppKit`+[NSDisplayCycle currentDisplayCycle]+0x63
              AppKit`-[NSWindow(NSDisplayCycle)
_postWindowNeedsDisplayUnlessPostingDisabled]+0x121
              AppKit`-[NSWindow _setNeedsDisplayInRect:]+0x153
               18

              libsystem_kernel.dylib`__workq_kernreturn+0xa
              libsystem_pthread.dylib`start_wqthread+0xf
               34
```

Example


```
user@mac:~$ sudo dtrace -n 'python*:::function-entry {
>     printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
> }'

dtrace: description 'python*:::function-entry ' matched 1 probe

CPU     ID                    FUNCTION:NAME
  2  14174 PyEval_EvalFrameEx:function-entry utf_8.py:decode:15

  2  14174 PyEval_EvalFrameEx:function-entry <stdin>:<module>:1

  2  14174 PyEval_EvalFrameEx:function-entry pydoc.py:__call__:1793

  2  14174 PyEval_EvalFrameEx:function-entry pydoc.py:help:1832
```

How it works

- **Heavy lifting in the kernel**
  - Probe registration
  - D-code evaluation
    - Yep, there is a D VM in the kernel
  - Hooks setup and handling
- **Details in the user land**
  - Parse kernel and user land symbols
  - Compile D-code
  - Print data returned by the kernel
  - Register static probes defined by user land applications

Static probes

- **Defined at compilation time**
  - User → Info in the DOF (DTrace Object Format) section
    - Sent by dyld *Loader::registerDOFs* function at loading
  - Kernel → Info in SDT (Statically Defined Tracing) section
    - Parsed by *sdt_load_machsect*
- **NOP as a placeholder**
  - Replaced with a trap/lock when needed
- **Low overhead if the probe is not used**
  - Probe is not called but arguments are still computed
  - Possible to check if the probe is active to mitigate this

Static probes

user@mac:/tmp/$ cat example.d # from man dtrace

```
provider Example {
    probe increment(int);
};
```

user@mac:/tmp/$ dtrace -h -s example.d # generates example.h

user@mac:/tmp/$ cat example.c

```
#include <unistd.h>
#include "example.h"

int main() {
	int i = 0;
	while (1) {
		EXAMPLE_INCREMENT(i++);
		sleep(1);
	}
	return 0;
}
```

user@mac:/tmp/$ gcc -O3 example.c -o example

Under IDA

```
_main proc near
push    rbp
mov     rbp, rsp
push    rbx
push    rax
xor     edi, edi
nop     dword ptr [rax+rax+00000000h]
```

```
loc_100003DB0:
lea     ebx, [rdi+1]
nop
nop     dword ptr [rax+00h]
mov     edi, 1          ; unsigned int
call    _sleep
mov     edi, ebx
jmp     short loc_100003DB0
_main endp

__text ends
```

Static probes


```
user@mac:/tmp/$ ./example &
[1] 9062
user@mac:/tmp/$ sudo dtrace -l -P 'Example*'
   ID   PROVIDER      MODULE                          FUNCTION NAME
14174 Example9062     example                              main increment
user@mac:/tmp/$ sudo dtrace  -n 'Example*::: {print(arg0);}'
dtrace: description 'Example*::: ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  14174                   main:increment int64_t 0x39
  0  14174                   main:increment int64_t 0x3a
```

Under lldb


```
user@mac:~$ lldb -p 9062
[...]
(lldb) disassemble -n main
example`main:
    0x109bdfda0 <+0>:  push   rbp
    0x109bdfda1 <+1>:  mov    rbp, rsp
    0x109bdfda4 <+4>:  push   rbx
    0x109bdfda5 <+5>:  push   rax
    0x109bdfda6 <+6>:  xor    edi, edi
    0x109bdfda8 <+8>:  nop    dword ptr [rax + rax]
    0x109bdfdb0 <+16>: lea    ebx, [rdi + 0x1]
    0x109bdfdb3 <+19>: int3
    0x109bdfdb4 <+20>: nop    dword ptr [rax]
    0x109bdfdb8 <+24>: mov    edi, 0x1
    0x109bdfdbd <+29>: call   0x109bdfdc6               ; symbol stub for: sleep
    0x109bdfdc2 <+34>: mov    edi, ebx
    0x109bdfdc4 <+36>: jmp    0x109bdfdb0               ; <+16>
```

Dynamic probes

- **How they work**
  - Available symbols are used to find addresses
  - Original code is patched
  - Without instrumentation: 0 overhead
- **Problems**
  - No symbol
    - Static functions, stripped binaries, etc.
  - Inlined functions
  - Races
  - Return location

No symbol / inlined functions

- **In userland: use arbitrary addresses**
  - Special module “a.out”
  - Special function “-”
  - Ex: dtrace -n 'pid1:a.out:-:7fff201a800f {print(uregs[R_RAX])}’

- **Impossible in kernel land :’(**

Races

- **How can we restore the code without missing probe hits?**
  - Don't restore it!
- **In user land**
  - Emulate simple / frequent instructions and branches
  - Relocate all the others in a scratch buffer and redirect execution
- **In kernel land**
  - Emulate the instruction
  - Only support a few instructions
    - Prologue and epilogues
  - Thus the impossibility to place probes at arbitrary addresses

Relocation is not that easy

“There's a further complication in 64-bit mode due to %rip-relative addressing. While this is clearly a beneficial architectural decision for position independent code, it's hard not to see it as a personal attack against the pid provider since before there was a relatively small set of instructions to emulate”

A comment in dev/i386/fasttrap_isa.c

Return location

- **Symbols give you functions start**
  - Return instructions might be anywhere in the function
  - Tail call optimisations… no return instructions
  - Data in code (jump tables, strings etc.)
- **Dtrace needs to disassemble the functions**
  - There are full disassemblers in the kernel!
  - Basic heuristics to detect jumps tables
  - See *fbt_provide_probe* and *dt_pid_create_return_probe* for the various architectures

Disassembly is not that easy

“This is horrible. Some SIMD instructions take the form 0x0F 0x?? ..., which is easily decoded using the existing tables. Other SIMD instructions use various prefix bytes to overload existing instructions. For Example, addps is F0, 58, whereas addss is F3 (repz), F0, 58. Presumably someone got a raise for this.”

A comment in bsd/dev/i386/dis_tables.c

Types

- **All kernel structures can be used in D scripts**
  - Even those not documented/released by Apple

- **Function arguments are known**
  - Use the args array and not argX

Types


```
user@mac:~$ sudo dtrace -n 'fbt::read:entry {
>     printf("pid: %d\n", args[0]->p_pid);
>     print(args[0]->p_ucred->cr_posix);
> }'

CPU     ID                    FUNCTION:NAME
  1 139880                   read:entry pid: 420
struct posix_cred {
    uid_t cr_uid = 0x1f5
    uid_t cr_ruid = 0x1f5
    uid_t cr_svuid = 0x1f5
    short cr_ngroups = 0x10
    gid_t [16] cr_groups = [ 0x14, 0xc, 0x3d, 0x4f, 0x50, 0x51, 0x62, 0x2bd,
0x21, 0x64, 0xcc, 0xfa, 0x18b, 0x18e, 0x18f, 0x190 ]
    gid_t cr_rgid = 0x14
    gid_t cr_svgid = 0x14
    uid_t cr_gmuid = 0x1f5
    int cr_flags = 0x2
}
```

Types

- **ctfdump can extract all the type information**
  - Useful when Apple delays in releasing sources

user@mac:~$ ctfdump /System/Library/Kernels/kernel | grep 'STRUCT proc ' -A 5

```
<306> STRUCT proc (1216 bytes)
  p_list type=115 off=0
  task type=76 off=128
  p_pptr type=307 off=192
  p_ppid type=56 off=256
  p_original_ppid type=56 off=288
```

Limitations

- **Root only**
- **DTrace is restricted under SIP**
  - Only syscall/mach_trap/profiles probes
  - No RW on CS_RESTRICT processes
    - all entitled processes
- **Kernel access is restricted**
  - No arbitrary probes
  - No RW on registers
  - No memory write

- **Can deny DTrace usage**
  - ptrace(PT_DENY_ATTACH);
  - Used by Apple in DRM
- **D is really limited**
  - No loops, no ifs...
- **Probes are not well documented**
  - Read the source, Luke

Detection

- **Not that easy as DTrace is in the kernel**
- **PT_DENY_ATTACH easy to bypass**
  - proc→p_lflag &= ~P_LNOATTACH
- **Classic anti-dbg tricks**
  - Code integrity
  - Detect int3 at API entries
  - Time sensitive functions
- **Specific DTrace detection**
  - Search anonymous executable pages

Conclusion

- **DTrace is a powerful tool…**
  - Especially to debug performance problems
  - But not only
- **… but it can be frustrating**
  - D…
  - Kernel restrictions
- **Future work: extend it?**
  - Code base is really clean and modular

https://www.linkedin.com/company/synacktiv
https://twitter.com/synacktiv
View all our publications on https://synacktiv.com