Extended Berkeley Packet Filter

Yannick Chevalier
Université de Toulouse
CSA M1, Security

PLAN

CONTEXT

BPFTRACE

CONCLUSION

PROGRAMS

PROGRAM FILE

- An object file contains the instructions, as well as functions' and static data addresses
- All addresses are stored in a symbol table
- Undefined functions have to be found by the system at execution time

Dynamic libraries *vs* Static libraries

- `objdump -T prog` shows the symbol table of a program

```
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 freeaddrinfo
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.3.4 __sprintf_chk
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 socket
000000000048eb80 g    DF .text 00000000000000b3  Base        camlBiblio__cut_after_nth_rec_1221
00000000004702f0 g    DF .text 0000000000000059  Base        camlConstraints__fun_1645
000000000047a610 g    DF .text 000000000000002f  Base        camlUnif_AC__fun_3543
000000000049fac0 g    DF .text 0000000000000122  Base        camlHashtbl__remove_1185
00000000006d77a0 g    D  .data 0000000000000000  Base        caml_exn_Stack_overflow
```

PROCESS EXECUTION

OPERATING SYSTEM'S JOB

1. Load a program file in memory
2. Provide virtual addresses as well as real addresses (in memory)
3. Map virtual addresses to real addresses

   Partial mapping, hence Segfaut
4. (Optionally) add a random offset for the base addresses of functions
5. Start execution at address 0 of the text
6. Initialisation is provided by the RunTime library (*e.g.* libcrt), which calls `main`

Process Execution

Operating system's job

1. Load a program file in memory
2. Provide virtual addresses as well as real addresses (in memory)
3. Map virtual addresses to real addresses

   Partial mapping, hence Segfaut

4. (Optionally) add a random offset for the base addresses of functions
5. Start execution at address 0 of the text
6. Initialisation is provided by the RunTime library (*e.g.* libcrt), which calls `main`

Process Execution

Operating system's job

1. Load a program file in memory
2. Provide virtual addresses as well as real addresses (in memory)
3. Map virtual addresses to real addresses

   Partial mapping, hence Segfaut
4. (Optionally) add a random offset for the base addresses of functions
5. Start execution at address 0 of the text
6. Initialisation is provided by the RunTime library (*e.g.* libcrt), which calls `main`

Process Execution

Operating system's job

1. Load a program file in memory
2. Provide virtual addresses as well as real addresses (in memory)
3. Map virtual addresses to real addresses

   Partial mapping, hence Segfaut

4. (Optionally) add a random offset for the base addresses of functions
5. Start execution at address 0 of the text
6. Initialisation is provided by the RunTime library (*e.g.* libcrt), which calls `main`

PROCESS EXECUTION

OPERATING SYSTEM'S JOB

1. Load a program file in memory
2. Provide virtual addresses as well as real addresses (in memory)
3. Map virtual addresses to real addresses

   Partial mapping, hence Segfaut
4. (Optionally) add a random offset for the base addresses of functions
5. Start execution at address 0 of the text
6. Initialisation is provided by the RunTime library (*e.g.* libcrt), which calls `main`

PROCESS EXECUTION

OPERATING SYSTEM'S JOB

1. Load a program file in memory
2. Provide virtual addresses as well as real addresses (in memory)
3. Map virtual addresses to real addresses

   Partial mapping, hence Segfaut
4. (Optionally) add a random offset for the base addresses of functions
5. Start execution at address 0 of the text
6. Initialisation is provided by the RunTime library (*e.g.* libcrt), which calls `main`

PROGRAM MODIFICATION

KERNEL-SIDE

1. The OS can replace calls to functions by other calls
2. The OS can insert and delete in memory instructions at any place

USER-SIDE

1. Needs to be root, have the `SYS_PTRACE` capability (Linux), or own the process

   *BSD : for non-root users, must be the parent of the process

2. The ptrace call allows for the reading and writing at runtime of a process memory (and of the registers)

PROGRAM MODIFICATION

KERNEL-SIDE

1. The OS can replace calls to functions by other calls
2. The OS can insert and delete in memory instructions at any place

USER-SIDE

1. Needs to be root, have the `SYS_PTRACE` capability (Linux), or own the process

   *BSD : for non-root users, must be the parent of the process
2. The ptrace call allows for the reading and writing at runtime of a process memory (and of the registers)

Program Modification

Kernel-side

1. The OS can replace calls to functions by other calls
2. The OS can insert and delete in memory instructions at any place

User-side

1. Needs to be root, have the `SYS_PTRACE` capability (Linux), or own the process

   *BSD : for non-root users, must be the parent of the process

2. The ptrace call allows for the reading and writing at runtime of a process memory (and of the registers)

Program Modification

Kernel-side

1. The OS can replace calls to functions by other calls
2. The OS can insert and delete in memory instructions at any place

User-side

1. Needs to be root, have the `SYS_PTRACE` capability (Linux), or own the process

   *BSD : for non-root users, must be the parent of the process
2. The ptrace call allows for the reading and writing at runtime of a process memory (and of the registers)

eBPF

The language

- Bytecode language
- Can be interpreted (like Java) by a Virtual Machine
- Can be compiled at run time (like JavaScript) into Machine Code

Kernel Support

- An eBPF JIT compiler
- A sandbox (with restricted memory access for spatial separation) for the execution of the compiled code
- A verifier that verifies the time separation of the code
- A module that receives commands from Userland to perform runtime modification
  - On users' code (uprobe)
  - On kernel's code (kprobe)
- Communication : shared memory or file

PLAN

CONTEXT

BPFTRACE

- BPFTrace snippet
- BPFTrace probes
- BPFTrace programs

CONCLUSION

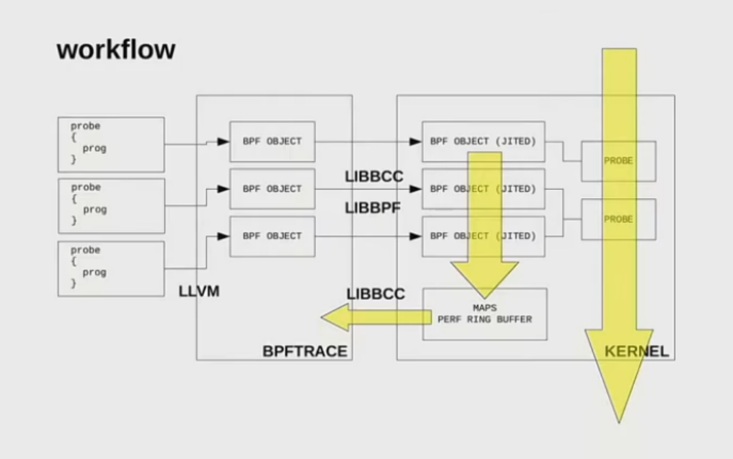
OUTLINE

BPFTRACE

- BPFTrace snippet
- BPFTrace probes
- BPFTrace programs

PRESENTATION

FROM BPFTRACE INTERNALS, JIRI OLSA, 2020



The eBPF Bytecode

Environment	
r<1-10>	registers
r<1-5>	func arguments
r0	return value
r10	stack on entry
r1	context on entry
map[id :X]	map descriptor

Remarks

- Maps are associative tables
- Maps are the effective way to store information
- Possible calls to helper functions
- Very limited stack size (512b)

USING MAPS (1/2)

USING A C INTERFACE

MAP UPDATE

```
int (*bpf_map_update_elem ) (
            void * map ,
            const void * key ,
            const void * value ,
            uint64_t flags ) ;
```

REMARKS

- Most maps are hashtables (*i.e.* _htab_map_update_elem or _htab_percpu_map_update_elem)
- Translation is provided for the supported languages !
- Map Ids (the pointer map) are shared between calls

USING MAPS (2/2)

USING A C INTERFACE

MAP LOOKUP

```
const void *
(*bpf_map_lookup_elem ) (
                void * map ,
                const void * key ) ;
```

REMARKS

- returns NULL if the key is not in the map
- returns the stored value (of type **const void** *)

MAPS FROM BPFTRACE

BPFTrace

@, @[name(,name)*]	default map, map name
count(name)	number of elements

Variables

Naming

```
@name = 0;
```

Variables use	
@name	Global variable name of type int
$name	Per-event variable name of type int
toto, gcc	string constants/names
arithmetics	as in C
printf	as in C
min, max, avg, sum, stats, hist, lhist	aggregates on all calls using internally a map

PRE-DEFINED VARIABLES (1/2)

Variable Name	Meaning
pid	Process ID (kernel tgid)
tid	Thread ID (kernel pid)
uid	User ID
gid	Group ID
nsecs	Nanosecond timestamp
elapsed	Nanoseconds since bpftrace initialization
cpu	Processor ID
comm	Process name
kstack	Kernel stack trace
ustack	User stack trace

PRE-DEFINED VARIABLES (2/2)

Variable Name	Meaning
arg0,. . ., argN.	Arguments to the traced function ; assumed to be 64 bits wide
sarg0, . . ., sargN.	Arguments to the traced function (for programs that store arguments on the stack) ; assumed to be 64 bits wide
retval	Return value from traced function
func	Name of the traced function
probe	Full name of the probe
curtask	Current task struct as a u64
rand	Random number as a u32
cgroup	Cgroup ID of the current process
cpid	Child pid(u32), only valid with the -c command flag
$1, $2, . . . , $N, $#.	Positional parameters for the bpftrace program

OUTPUT

Function Name	Usage
printf	prints at each event
print(name)	prints the map
hist(name)	histogram (power of two)
lhist(name,min,max,step)	histogram (linear)

CALLING EXTERNAL PROGRAMMS

BUILT-IN FUNCTION SYSTEM

- Argument : printf-like format string
- Evaluates the string as a program to call
- Needs an extra `–unsafe` flag

OUTLINE

BPFTRACE

BPFTrace snippet

BPFTrace probes

BPFTrace programs

PROBE

DEFINITION

- Location in a program where additional code has to be executed
- Can be either in the kernel or a position in the code of the program

Probe type	Usage
kprobe/kretprobe	Kernel function tracing
uprobe/uretprobe	Programs function tracing
tracepoint	Essentially system calls tracing
usdt	Tracing of statically defined tracepoints
interval	Time events (auxiliary)
software	Kernel software events
hardware	HW events (cache miss, etc.)

Hardware events :

cpu-cycles or cycles instructions cache-references cache-misses branch-instructions or branches branch-misses bus-cycles frontend-stalls backend-stalls ref-cycles

HARDWARE EVENTS

Name	Raised when
cpu-cycles or cycles	
instructions	
cache-references	
cache-misses	
branch-instructions or branches	
branch-misses	
bus-cycles	
frontend-stalls	
backend-stalls	
ref-cycles	

Software Events

Name	Raised when
cpu-clock or cpu	
task-clock	
page-faults or faults	
context-switches or cs	
cpu-migrations	
minor-faults	
major-faults	
alignment-faults	
emulation-faults	
dummy	
bpf-output	

Userland Probes

Syntax	Example
uprobe :library_name :function_name[+offset]	path to a library and relative address from a function start
uprobe :library_name :address	path to a library and absolute address in text
uprobe :path :function_name[+offset]	path to an object file and relative address from a function start
uprobe :path :address	path to an object file and absolute address in text

Call	Automatic variables
uprobe	arguments arg0, arg1, . . . , argN
uretprobe	return value in retval

Userland Probes

Syntax	Example
uprobe :library_name :function_name[+offset]	path to a library and relative address from a function start
uprobe :library_name :address	path to a library and absolute address in text
uprobe :path :function_name[+offset]	path to an object file and relative address from a function start
uprobe :path :address	path to an object file and absolute address in text

Examples

One can get the files opened or the user input in terminals with :

```
uprobe:/lib/x86_64-linux-gnu/libc-2.31.so:fopen { \
                    printf("file␣opened:␣\"%s\"\n", str(arg0)
uretprobe:/bin/bash:readline { \
                    printf("readline: \"%s\"\n", str(retval))
```

KERNEL PROBES

Syntax	Example
kprobe :function_name[+offset]	Kernel function when called, with relative address
kretprobe :function_name	Kernel function return value

Call	Automatic variables
kprobe	arguments arg0, arg1, . . . , argN
kretprobe	return value in retval

Kernel Probes

Syntax	Example
kprobe :function_name[+offset]	Kernel function when called, with relative address
kretprobe :function_name	Kernel function return value

Examples

One can get all the files opened on the system with :

```
kprobe:do_sys_open { printf("opening:␣%s\n", str(arg1)); }'
```

TRACEPOINTS

MOSTLY FOR SYSTEM CALLS

Syntax	Called when
tracepoint :syscalls :sys_enter_name	When a program makes the name system call
tracepoint :syscalls :sys_exit_name	When the system call name returns

Type	Called when
enter	pid making the call, and the arguments
exit	pid to which the value is returned, and the returned value (dependent on each system call)

WHICH VALUES ARE AVAILABLE ?

```
cat /sys/kernel/debug/tracing/events/syscalls/\
        sys_enter_open/format
```

```
name: sys_enter_openat
ID: 608
format:
field:unsigned short common_type; offset:0; size:2; signed:0;
field:unsigned char common_flags; offset:2; size:1; signed:0;
...
```

TIME EVENTS (INTERVAL)

GOAL

- “Synthetic” event to perform something periodically
- syntax : `interval:time`
- time is in microseconds (us), milliseconds (ms), seconds (s), or every *n* per second (Hz)
- Normally used with another probe, and with the two probes sharing (at least) a global variable

OUTLINE

BPFTRACE

- BPFTrace snippet
- BPFTrace probes
- BPFTrace programs

BPFTrace Programs

Structure

- A BPFTrace programm attaches code snippets with conditions to probes (see examples above)

  Conditionals can use the automatic variables, are in / . . . / between the probe and the code

- Two additional probes :

  BEGIN : to initialise maps and data, code executed before all other code

  END : code executed when exiting the program, useful for printing stats

Same as Awk programs !

Calling BPFTrace

```
bpftrace -e 'kprobe:do_nanosleep␣\
␣␣␣␣␣␣␣␣␣␣/tid␣==␣1234/␣{␣printf("sleep␣by␣%d\n",␣tid);␣}'
```

PLAN

CONTEXT

BPFTRACE

CONCLUSION

TO DELVE FURTHER...

> **G^AL K. ALEXANDER**
>
> The difference between you and us is that we know what runs on a system

MAN BPF-HELPERS : not always current list of functions

KERNEL CODE : headers or src

- include/uapi/linux/bpf.h : all heper functions
- net/core/filter.c : network related functions
- kernel/trace/bpf_trace.c : tracing functions
- kernel/bpf/ : other functions (cgroups,...)

BPFTOOL : bpftool feature probe gives the name of existing functionalities in the running kernel

TO DELVE FURTHER...

GAL K. ALEXANDER

The difference between you and us is that we know what runs on a system