Sécurité des logiciels

Dynamic/static analysis

Samuel Thibault <samuel.thibault@u-bordeaux.fr>

Static vs Dynamic analysis

Reminder: Halting problem

Both are useful

- Static analysis
  - Issues that don't seem to pop up in practice
  - Fix bugs before seeing only their symptoms
- Dynamic analysis
  - Issues that pop up in practice and can't be analyzed

Dynamic analysis

Dynamic analysis

- We discussed compiler-injected checks last week
- Here, we start from already-compiled binaries
  - Valgrind
- First, let's discuss emulation vs simulation

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- Emulator

```
readmem(&opcode, 1, r[EIP]);
switch (opcode) {
case INS_INCL:
  readmem(&args, 1, r[EIP]+1);
  r[args&7]++;
  break;
case INS_MOVL:
  readmem(&args, 1, r[EIP]+1);
  ... readmem(&val, 4, r[EIP]+2);
  ... r[args&7] = val;
  break;
case INS_JMP:
  readmem(&target, 4, r[EIP]+1);
  r[EIP] = target;
  break;
```

Emulation vs simulation

So emulation is really reproducing CPU behavior

- 1-for-1
- Nachos: mipssim.cc
- Also called “interpreter”

Actually… That is what happens inside the CPU :)

- See microcode/microprogramming

In software this is sssllooowwwww

Emulation vs simulation

Simulation

- Just let original program run on the native CPU
  - At full speed!
- This is actually what gdb does
  - Also kvm, xen, User-Mode Linux, ...
- Catch situations as needed, e.g. breakpoint
  - Can tell the processor to stop at a given address
  - Or use the **`INT3`** instruction (fits in one byte! **`\xCC`**)
  - Or tell the Operating System to catch system calls

- Very fast
- But cannot observe code behavior at fine grain

Emulation vs simulation

What about getting the best of both worlds?

- Running native code at full speed
- But that does what we want

Just-In-Time compilation (JIT)

- Produce native code that does what we want, on the fly
- Just let it execute
- Cache the produced code, for subsequent executions

Emulation vs simulation

- Real program

```
incl %ebx
movl $1,%eax
jmp somewhere
```

- JIT version

```
EAX: .long 0
EBX: .long 0
[...]

movl EBX,%eax
incl %eax
movl %eax,EBX

movl $1,%eax
movl %eax,EAX

movl $somewhere,%eax
jmp DO_JUMP
```

Emulation vs simulation

JIT

- Basically like a compiler
  - A C compiler takes a C source code and produces assembly code
  - A JIT takes a binary code and produces binary code
- Stores emulated CPU state in memory
  - Variables EAX, EBX, etc. for registers

- But optimizes emulated register access
  - Native register allocation for emulated registers
  - Register spilling whenever needed, just like compiler

```
addl %eax,%ebx

incl %ebx
```

```
movl EBX,%eax
addl EAX,%eax
incl %eax
movl %eax,EBX
```

Emulation vs simulation

JIT

- Significantly longer code
- But still way faster than pure emulation

- Can inject whatever check we want, e.g.
  - addresses
  - undefined values
  - ...

Valgrind

JIT

- Reads blocks of code from your binary
  - Separated by jumps / syscall
  - Typically the content of
    - Blocks of function with no if/for/while
    - if, for, while
- Produces corresponding JIT emulation blocks
  - With pluggable additional checks
    - memcheck: correct memory accesses
    - helgrind: correct concurrency
    - cachegrind: cache efficiency analysis
    - callgrind: call graph analysis
    - massif: memory usage efficiency analysis
  - Caching

Valgrind, a trivial example

- Real program

```
incl myvar
```

- Valgrind version

```
pushl $4
pushl $myvar
call check_data_rw
incl myvar
```

memcheck, what it does

Traps calls to malloc/free

Replaces them with its own implementations:

- malloc allocates a fresh area
- free checks it is a valid pointer
- free **does not** deallocate
- free marks area as invalid (thus forever)

Then, on memory access, check that either

- Address is on the stack
  - Note: hard to determine which parts of the stack is valid
- Address is in the data segment
  - Note: hard to determine which parts of the data segment is valid
- Address is **inside** a malloc-ed area
  - That one, however, is precise

memcheck, how it does

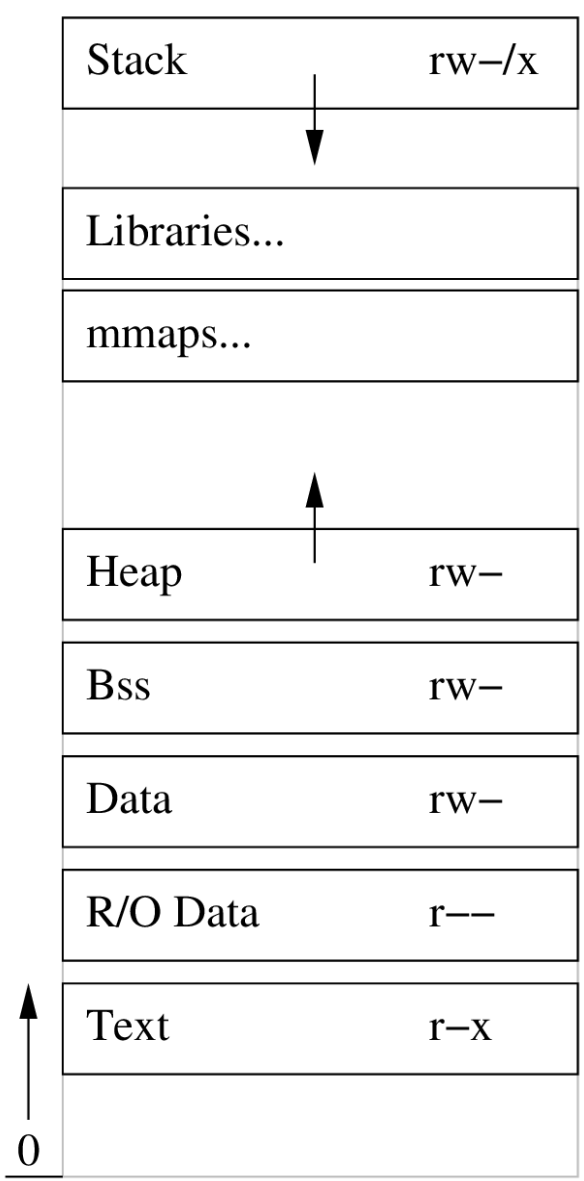
Checking against list of allocations would be terribly costly

→ Maintains an “A” array

- One bit per byte of memory of the process
  - 1 if valid address, 0 if invalid address
- Sparse
  - Collapse large chunks of contiguous 1 / 0

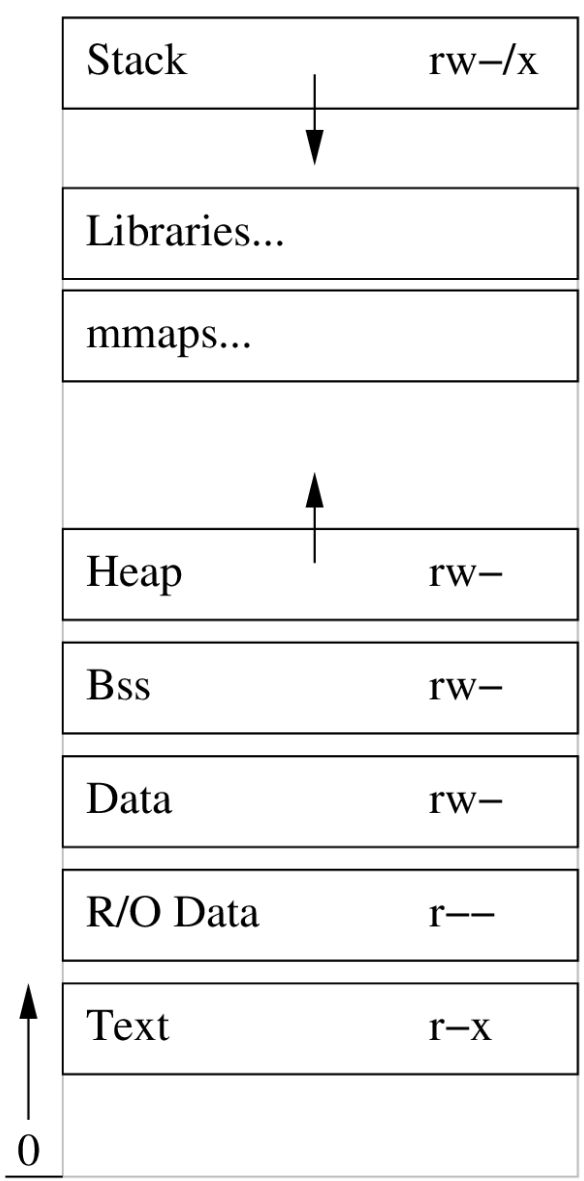
- At program startup, mark bss/data/rodata segments as valid
- At stack allocation, mark allocated piece as valid
- At stack deallocation, mark deallocated piece as invalid
- On malloc, fill the bits for the allocated area
- On free, clear the bits for the allocated area

memcheck, how it does




```
11111111
11111111
11100000
00000000
00000000
11111111
11111111
11111111
11111111
11111111
11111111
00000000
00000000
00000000
00111111
11111111
11111111
00000000
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
00000000
00000000
00000000
```

memcheck, how it does




```
[1]
[1]
11100000
[0]
[0]
[1]
[1]
[1]
[1]
[1]
[0]
[0]
[0]
00111111
[1]
[1]
[0]
[1]
[1]
[1]
[1]
[1]
[1]
[1]
[1]
[1]
[1]
[1]
[1]
```

memcheck, what it does (2)

Tracks uninitialized values

→ Maintains a “V” array

- One bit per bit of memory of the process!!
  - 0 if initialized bit, 1 if uninitialized bit
- Sparse

- At program startup, mark bss/data/rodata as initialized
- At stack allocation / malloc, mark new area as uninitialized
- On initializing memory, mark area as initialized

memcheck, what it does (2)

Tracks uninitialized values

Note: check is done **only** when actually used

- Avoids a huge lot of false positives

```
Conditional jump depends on uninitialised value(s)
```

I.e. when copying data from a variable to another, just **copy** over the V bits.

I.e. sometimes hard to determine where the initialization is missing :/

helgrind, what it does

Tracks concurrency tricks

e.g. traps calls to pthread_mutex_lock/unlock

- Checks lock acquisition ordering
- Checks no memory access at same address without a lock held

valgrind: notes

Valgrind runs over the **whole** user process

- Including libraries, e.g. libc
- It may warn about bugs in libc :/
- It may warn about code in libc, but error is in your code

It does not run over the kernel code

- It lets system calls be made natively
- But it checks parameters, and marks memory accordingly

valgrind: notes

- False positives
- Bugs in libraries

→ “suppressions”

  - Rules in suppression file
  - **`VALGRIND_HG_DISABLE_CHECKING(variable)`**

- home-made allocator
  - **`VALGRIND_MALLOCLIKE_BLOCK()`**

Static analysis

Static analysis

Compilers can do a lot of trivial checks

- Function parameters typecheck
  - That is **why** we #include . h files

```
ssize_t sendfile(int out, int in, off_t *offset,
size_t count);

int my_offset;
sendfile(out, in, &my_offset, 10);
```

    warns, and indeed might overflow!

- Also returned type!
  - Without prototype, return type assumed to be int
    - Bogus if it was actually a pointer!!

Static analysis

Compilers can do more involved checks

Function annotations

- char *strcat(char *dst, const char *src)
  __attribute__((access(read_write, 1), access(read_only, 2)));
  - Compiler will know that dst needs to be initialized somehow
- int my_printf(void *foo, const char *fmt, ...)
  __attribute__((format (printf, 2, 3)));
  - Compiler will check parameters according to printf-like format.
- char *strcpy (char *dest, const char *src)
  __attribute__((nonnull(1, 2)))
  - Compiler will check parameters are not NULL
- ...

Static analysis

With optimization enabled, compiler can go further

Interval analysis

- ```
  int t[10];
  for (int i = 0; i <= 10; i++)
    t[i]=1;
  ```

- ```
  if (i > 10)
    return;
  t[i] = 1;
  ```

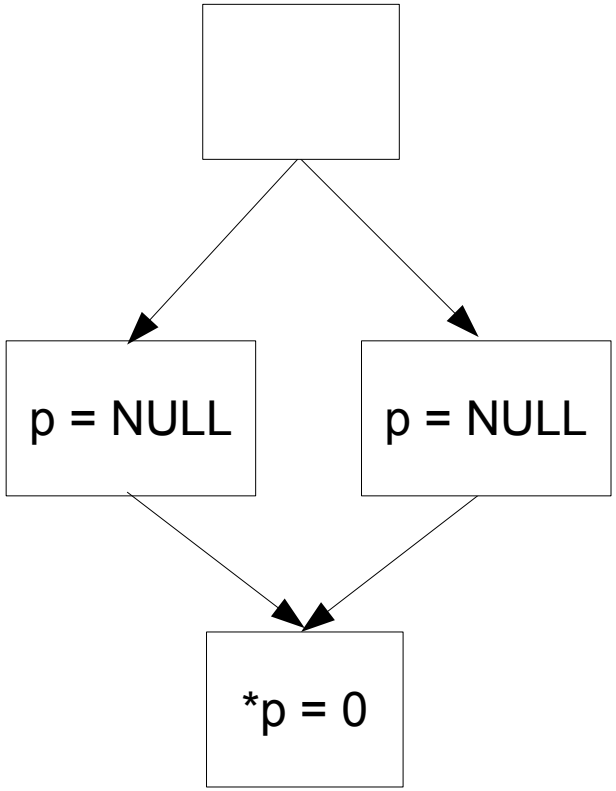
- ```
  if (p == NULL)
    printf("oops?\n");
  *p = 0;
  ```
  ```

Static analysis

With optimization enabled, compiler can go further

Inter-block analysis

```
if (i == 0) {
  printf("foo\n");
  p = NULL;
} else {
  printf("bar\n");
  p = NULL;
}
*p = 0;
```



Static analysis

With optimization enabled, compiler can go further

Inter-procedural analysis

```
static void f(int *p) {
  *p = 0;
}
static void g(void) {
  int *q;
  f(q);
}
```

Static analysis

With optimization enabled, compiler can go further

- gcc does some of it
  - But not all it could, it's just a compiler
- cppcheck does more of it
- coverity does a lot more of it

We'll see that in the practice lesson

Stressing your code : fuzzing, coverage, CI

Fuzzing

Making sure your program doesn't misbehave

- Just feed it random stuff!
  - **`unzip /dev/random`**
- Or not so random
  - Prepare zip-looking **`file.zip`**
  - **`unzip file.zip`**
- Even better, carefully-chosen random
  - Prepare zip-looking **`file.zip`**
  - **`unzip file.zip`**
  - Observe which parts of unzip have been executed
  - Try to modify **`file.zip`** randomly
  - Observe again
  - etc. until all parts of unzip have been tried

→ Code **coverage**

Fuzzing tools : AFL, libFuzzer, honggfuzz

Code coverage

Not only fuzzing :)

Program testsuite

- Should check all parts of the program
- **`gcc --coverage`**
- **`gcov`** generates coverage report

Continuous Integration

All of this (valgrind, [altu]san, cppcheck, coverity, fuzz) take time

But it can be all automated!

Run this during the night
→ Morning report of all the bugs you commited the day before

Run this on merge requests
→ Before integrating external contributions

Conclusion

Conclusion

Valgrind has extremely little false positives

- Always keep your code valgrind-warnings-free
- Use CI for this

asan/lsan/usan have extremely little false positives

- Can as well just develop with asan always enabled

Static analysis tools are costly

- Use CI for them