Reverse Engineering

Dr Remi Géraud-Stewart, École normale supérieure

9 oct 2019

Quick review

- Reverse engineering: why and what for? Legal framework
- What we're looking for in RE
- Compilation, ABI, call conventions, disassembly
- Static analysis: ghidra, galileo, CFG reconstruction, type inference

`https://github.com/5d54626cc43ba7ce625903ffa39c049c/RV`

Today

- Dynamic analysis
- Side-channels and fault injection
- Practical considerations

Why 'dynamic' analysis?

Why 'dynamic' analysis?

- RE is indecidable

Why 'dynamic' analysis?

- RE is indecidable
- Static analysis on large programs is impractical

Why 'dynamic' analysis?

- RE is indecidable
- Static analysis on large programs is impractical
- Some programs cannot be statically analysed

Why 'dynamic' analysis?

- RE is indecidable
- Static analysis on large programs is impractical
- Some programs cannot be statically analysed
- Some programs have JITs, memory encryption, obfuscation...

Why 'dynamic' analysis?

- RE is indecidable
- Static analysis on large programs is impractical
- Some programs cannot be statically analysed
- Some programs have JITs, memory encryption, obfuscation…
- 'Dynamic': as the program 'runs'

Why 'dynamic' analysis?

- Static analysis finds properties that hold for all executions
- Dynamic analysis finds properties that hold of one or more executions
  - Can't prove a program satisfies a particular property
  - But can prove that it doesn't!
- In other terms, dynamic analysis is **more precise but less sound**

Example1: will it do the thing?

```
def collatz(x):
  if x == 1:
    return True
  elif x % 2 == 0:
    return collatz(x/2)
  else:
    return collatz(x*3 + 1)

if collatz(user_input):
   do_something()
```

Example2: Packers

- To hide their payload, many malware authors use a **packer**
- The packer is a program that uncompresses and runs the payload **at runtime**
- Instead of analysing statically the binary, we let the packer do the unpacking, decrypting, decompressing etc. into memory of the real payload.

Example3: Parsers

- A parser processes information in some interchange format (e.g. JSON)
- In practice, ensuring the correctness of realistic parsers is out of reach of static analysis (lack of specs, combinatorial explosion)
- What if we 'just' run the program with random inputs?

- We can see the program being executed
  - Understand its behaviour
  - Let it do the work for us, we focus on some part
  - Identify inputs that cause special events
  - Learn information about secret data
- We may modify the program as it runs
  - Change the CFG
  - Learn information about secret data

A few precautions

Running a program of unknown origin may have unintended consequences

Running tools on a program of unknown origin may have unintented consequences

Be careful, use a computer that you don't care about.

Frida

- Dynamic instrumentation framework
- `sudo pip install frida-tools`

Volatility

- Advanced memory forensics framework
- `sudo apt-get install volatility`

American Fuzzy Lop

- Instrumentation-driven fuzzer
- `sudo apt-get install afl`

angr

- Concolic analysis engine
- `pip install angr`

gdb

- Debugger
- `sudo apt-get install gdb`

Wireshark

- Network packet analyser
- `sudo apt-get install wireshark`

But first, a bit of theory

Debugging and Hooking

- OS provices callbacks to trace and interrupt a target's execution
- Linux `ptrace` allows controlling another process
- Neither very efficient nor stealthy, designed for developers
- Some debuggers allow backwards execution (time travelling) from a breakpoint

- `break main` (set a breakpoint at the beginning of the `main` function)
- `disass main` (show `main` assembly code)
- `r [arguments]` (run until first breakpoint is reached)
- `break *0x[address]` (set a breakpoint at this code address)
- `cont` (continue until the next breakpoint)
- `info reg` (show CPU registers)
- `x/32bx 0x[address]` (display contents of memory at this address)
- `x/s 0x[address]` (display string at this address)
- `bt` (backtrack function calls)

- An alternative approach is to intercept a target's library or system calls
- `LD_PRELOAD=/path/to/my/malloc.so /bin/ls`
- Much more efficient and stealthy
- Well-suited for protocol analysis
- Example: `wireshark`

Instrumentation

Instrumentation

Instrumentation: ”Insert additional instructions in a program”

- **Source-based instrumentation** AIMS, Paradyn, Pablo
  - Source code is modified, new instructions are added
  - Compile/link
  - Cannot handle dynamically-generated code (JIT etc)
  - Language dependent
- **Binary-only instrumentation** (see next slide)
  - Attach to running process
  - No need to recompile/link
  - Can handle dynamically-generated code (see next slide)
  - Architecture dependent

Source-based instrumentation is faster but malware rarely comes bundled with source code.

- **Static binary instrumentation** — ATOM, EEL, Morph
    - Insert instrumentation operations before running the target
    - 'As easy as' inserting instructions
    - May instrument whole program (= big overhead)
- **Dynamic binary instrumentation** — Valgrind, DynamoRIO, PIN
    - Turn on/off instrumentation, change it at runtime
    - Requires a dispatcher ('trampoline')
    - Can instrument selectively without relinking (= fast)

Dynamic instrumentation can apply to generated code (e.g. self-mutating programs) whereas static instrumentation does not.

Demo / Tutorial : Frida

Profiling

We can't instrument everything. Why?

We need an **instrumentation strategy**

- The 'right strategy' depends on what we're looking at
- Often, we are interested in the CFG: **calls, branches and jumps**
- In other cases (e.g. crypto) we are interested in **memory accesses** instead

```c
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;#\
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

```c
#include <stdio.h>
main(t,_,a) char *a; {
    if ((!0) < t)  {
        if (t < 3) main(-79, 13, a+main(-87, 1-_, main(-86, 0, a+1)+a)); // .1
        if (t < _) main(t+1, _, a); // .2
        main(-94, -27+t, a); // .3
        if (t == 2 && _ < 13)  main(2, _+1, ""); // .4
    }
    else if (t < 0) {
        if (t < 72) main(_, t, STRING_A); // .5
        else if (t < -50) {
            if (_ == *a) putchar(31[a]); // .6
            else main(-65, _, a+1); // .7
        }
        else main((*a == '/')+t, _, a+1); // .8
    }
    else if (0 < t) // .9
        main(2, 2, "%s");
    else if (*a != '/')  // .10
        main(0, main(-61, *a, STRING_B), a+1 );
}
```

- **Edge profiling**:
    - Count how many times each jump is taken
    - In practice does not work well
- **Path profiling**:
    - Count how many times a path is taken when running the program
    - Efficient algorithms exist [BL96]

Why is this interesting?

Path	Taken
9	1
1, 3, 4	1
1, 2, 3	1
1, 2, 3, 4	10
3	11
2, 3	55
5	114
10	2358
6	2358
8	24931
7	39652

When running the program, **2358 characters are printed**.

Path profiling helps us understand what part of a program does what [Bal99].

We can also go back at the paths and identify under what conditions they are followed (a la mano, or using an SMT solver such as Z3)

$\rightarrow$ path (1, 2, 3) is used if `t == 2 && t < _ && _ >= 13`

this gives even more information about the program's design!

Concolic analysis

Idea: combine **conc**rete execution with a symb**olic** solver.

- CUTE [SMA05], DART [GKS05], KLEE [CDE08]
- In practice, we have to ”concretize” some symbolic variables
- Trade-off between correctness, completeness and efficiency
- Extremely efficient bug-finding / vuln-confirming approach

```
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concolic execution: Example

```
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concrete	Symbolic	Condition
$x = 22, y = 7$	$x = x_0, y = y_0$	$\emptyset$

Concolic execution: Example

```
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concrete	Symbolic	Condition
$x = 22, y = 7$	$x = x_0, y = y_0$	$\emptyset$
$z = 14$	$z = 2 \times y_0$	$\emptyset$
$x = 22, y = 7$ $z = 14$	$x = x_0, y = y_0$ $z = 2y_0$	$2y_0 \neq x_0$

```c
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concrete	Symbolic	Condition
$x = 22, y = 7$	$x = x_0, y = y_0$	$\emptyset$
$z = 14$	$z = 2 \times y_0$	$\emptyset$
	Solve $2y_0 = x_0$	
$x = 22, y = 7$ $z = 14$	$x = x_0, y = y_0$ $z = 2y_0$	$2y_0 \neq x_0$

Concolic execution: Example

```
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concrete	Symbolic	Condition
$x = 22, y = 7$	$x = x_0, y = y_0$	$\emptyset$
$z = 14$	$z = 2 \times y_0$	$\emptyset$
x=2, y=1, z=2		$2y_0 = x_0$
$x = 22, y = 7$ $z = 14$	$x = x_0, y = y_0$ $z = 2y_0$	$2y_0 \neq x_0$

Concolic execution: Example

```
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concrete	Symbolic	Condition
$x = 22, y = 7$	$x = x_0, y = y_0$	$\emptyset$
$z = 14$	$z = 2 \times y_0$	$\emptyset$
x=2, y=1, z=2		$2y_0 = x_0$
	Solve $2y_0 = x_0$ and $x_0 > y_0 + 10$	
$x = 22, y = 7$ $z = 14$	$x = x_0, y = y_0$ $z = 2y_0$	$2y_0 \neq x_0$

Concolic execution: Example

```
int thing(int x) { return 2*x; }

void rainbow(int x, int y) {
    int z = thing(y);
    if (z == y) {
        if (x > y + 10) {
            kill_all_humans
        }
    }
}
```

Concrete	Symbolic	Condition
$x = 22, y = 7$	$x = x_0, y = y_0$	$\emptyset$
$z = 14$	$z = 2 \times y_0$	$\emptyset$
x=2, y=1, z=2		$2y_0 = x_0$
$x = 30, y = 15$ $z = 30$		$2y_0 = x_0$ $x_0 > y_0 + 10$
$x = 22, y = 7$ $z = 14$	$x = x_0, y = y_0$ $z = 2y_0$	$2y_0 \neq x_0$

Can I play with this?

- angr
  - Go on their website, follow the tutorials: `angr.io`
  - Integrates many of the state-of-the-art binary analysis techniques
  - Python-based
- KLEE
  - Go on their website, follow the tutorials: `klee.github.io`
  - A bit harder to install (advice: use the docker image)
  - Follow the gitbook: `verificaeconvalida.gitlab.io/gitbook-appunti/KLEE.html`
  - LLVM-based

CTF 'VeryAndroidoso' could be solved

- Using 'traditional' reverse engineering
- Using Frida (which works on Android)
- Using angr (which now works on Android)

`antoniobianchi.me/posts/ctf-defconquals2019-veryandroidoso/`

In 2016 and 2017, angr solved RE challenges *automatically*.

`p1kachu.pluggi.fr/writeup/re/2016/05/23/defconquals-baby-re-writeup/`

Fuzzing

Fuzzing: sending random inputs to a program, to explore its CFG

- This is very quickly inefficient if done stupidly
- **Coverage**: amount of branches explored by the fuzzer
- **Coverage-driven fuzzing**: fuzzer tries to explore all branches

But how does the fuzzer ’know’ that a branch has been taken?

Fuzzing: sending random inputs to a program, to explore its CFG

- This is very quickly inefficient if done stupidly
- **Coverage**: amount of branches explored by the fuzzer
- **Coverage-driven fuzzing**: fuzzer tries to explore all branches

But how does the fuzzer 'know' that a branch has been taken? **Instrumentation**

- `afl` provides an instrumenting compiler `afl-clang` or `afl-gcc`
- (you may also use your own intrumentation framework, e.g. PIN or DynamoRIO, etc.)
- Then runs the program on some valid input
- Then mutates this input and tries to find new branches.
    - Clever mutation strategy and heuristics make it fast [BPR16]
    - Tries to crash the program in new ways

- Get the source for some vulnerable program (e.g. old version of `binutils`)
- Compile with `afl-gcc` instead of `gcc` → `CC = afl-gcc ./configure`
- (LPT: deactivate OS crash handling: `echo core > /proc/sys/kernel/core_pattern`)
- Prepare legitimate input (e.g., an ELF file such as `ls`)
- Go: `afl-fuzz -i afl_input -o afl_output ./target -a @@`

Quick tutorial on afl

```
                 american fuzzy lop 1.74b (readelf)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 8 min, 24 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 1 min, 59 sec        │  total paths : 812    │
│ last uniq crash : 0 days, 0 hrs, 3 min, 17 sec        │ uniq crashes : 8      │
│  last uniq hang : 0 days, 0 hrs, 3 min, 23 sec        │   uniq hangs : 10     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 3158 (4.82%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.56 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 1 (0.12%)              │
│ stage execs : 295k/326k (90.31%)    │  new edges on : 318 (39.16%)           │
│ total execs : 552k                  │ total crashes : 63 (8 unique)          │
│  exec speed : 1114/sec              │   total hangs : 191 (10 unique)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 447/75.5k, 59/75.5k, 59/75.5k         │    levels : 2          │
│  byte flips : 7/9436, 0/5858, 6/5950                │   pending : 812        │
│ arithmetics : 0/0, 0/0, 0/0                         │  pend fav : 1          │
│  known ints : 0/0, 0/0, 0/0                         │ own finds : 811        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │  variable : 0          │
│        trim : 0.00%/1166, 38.39%                    ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 15%]
```

More about `afl`

- Can be scripted from python
- Can be used remotely
- Can be run in parallel
- Mainly used to find vulnerabilities in FOSS
- Bootcamp: `github.com/mykter/afl-training`

Memory analysis

Side-channel analysis

Fault injection

Thank you! Questions?

remi.geraud@ens.fr

References

[Bal99] Thomas Ball. “The Concept of Dynamic Analysis”. In: *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*. Ed. by Oscar Nierstrasz and Michel Lemoine. Vol. 1687. Lecture Notes in Computer Science. Springer, 1999, pp. 216–234. ISBN: 3-540-66538-2. DOI: `10.1007/3-540-48166-4\_14`. URL: `https://doi.org/10.1007/3-540-48166-4%5C_14`.

[BL96] Thomas Ball and James R. Larus. "Efficient Path Profiling". In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996*. Ed. by Stephen W. Melvin and Steve Beaty. ACM/IEEE Computer Society, 1996, pp. 46–57. ISBN: 0-8186-7641-8. DOI: `10.1109/MICRO.1996.566449`. URL: `https://doi.org/10.1109/MICRO.1996.566449`.

[BPR16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 1032–1043. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. ISBN: 978-1-931971-65-2. URL: `http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf`.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 213–223. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: a concolic unit testing engine for C”. In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. Ed. by Michel Wermelinger and Harald C. Gall. ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: `10.1145/1081706.1081750`. URL: `https://doi.org/10.1145/1081706.1081750`.