The Automated Exploitation Grand Challenge

A Five-Year Retrospective

Julien Vanegue

IEEE Security & Privacy Langsec Workshop

May 25th 2018

AEGC 2013/2018 vs DARPA Cyber Grand Challenge

- Was Automated Exploit Generation solved with DARPA CGC? Not quite.
- DARPA Cyber Grand Challenge ranked solutions on three criteria:
  1. Attack (how well you exploited)
  2. Defense (how well you defended against exploits)
  3. Performance (Availability of your services)
- CGC Post-mortem: "Cyber Grand Challenge: The Analysis" : `http://youtube.com/watch?v=SYYZjTx92KU`
- DARPA CGC scratched the surface, this presentation focuses on what is under the carpet.
- We focus on memory attacks and defense, there are other classes we dont cover here.

Automated Exploit Generation Challenges

Original AEGC 2013 challenges: http://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf

In a nutshell, attacks are decomposed into five classes:

CLASS 1: Exploit Specification ("sanitizer synthesis")
CLASS 2: Input Generation ("white-box fuzz testing")
CLASS 3: State Space Management ("combinatorial explosion")
CLASS 4: Primitive Composition ("exploit chaining")
CLASS 5: Information disclosure ("environment determination")

CLASS 1: Exploit Specification

For a given program p :
For all inputs $i_1, ..., i_n$:
For all assertions $a_1, ..., a_m$:

$$\textbf{Safety condition}: \forall a : \forall i : p(i) \Rightarrow a$$

$$\textbf{Attack condition}: \exists a : \exists i : p(i) \Rightarrow \neg a$$

where p is the program interpretation on the input i (for example, construction of a SMT formula)

CLASS 1 approach: Sanitizer synthesis

Sanitizers are developer tools to catch bugs early at run time:

- Valgrind (ElectricFence before it): heap sanitizer (Problem: too intrusive for exploit dev)
- Address Sanitizer: clang compiler support to solve same problem as Valgrind in LLVM.
- Cachegrind: simulate how program interacts with cache hierarchy and branch predictor.
- Helgrind: detect data races, locking issues and other thread API misuses.

Current research directions include coupling sanitizers with static analysis and/or symbolic execution.
See KLEE workshop talks: `https://srg.doc.ic.ac.uk/klee18`

CLASS 2: Input Generation

After defining what attack conditions are, input generation provides initial conditions to exercise sanitizing points:

- DART/SAGE: First white-box fuzzers (Godefroid, Molnar, Microsoft Research, 2006-)
- EXE/KLEE (Open-source Symbolic execution engine, Cadar, Dunbar and Engler, 2008-)
- American Fuzzy Lop aka AFL (Zalewski, 2014-) : (First?) open-source grey-box fuzzer
- More recently: Vuzzer, AFLfast, AFLgo, etc. (2016-)

These tools provide input mutation strategies to cover more path/locations in tested programs.
By now, input generation is a well-understood problem for restricted sequential programs.

CLASS 3: State-space management

A well known problem in program analysis is **Combinatorial explosion**. For several classes of programs, this leads to exponential blow-up of the state space:

- Multi-threaded programs: For i instructions, n threads: scheduling graph contains n^i states.
- Heap-based programs: For i allocations, n possible allocation size bins: heap config space contains n^i states.

Motivation: Data Only Attacks (DOA)

Data-only attacks form a vulnerability class that can bypass exploit protections such as:

- Non-execution mitigations (DEP, W$\wedge$E) : no code injection needed.
- Control-Flow Integrity (CFI) : no code redirection needed.

Under certain conditions, it can defeat:

- Address Space Layout Randomization (if it does not rely on absolute addresses)
- Heap meta-data protections (if it does not rely on heap meta-data corruptions)

Example of DOA: heartbleed (lines up chunks in memory to leak private material)

Decide safety using Adjacency predicate

$$\forall x \neg \exists y : TGT(y) \wedge ADJ(x, y) \wedge OOB(x)$$

- ADJ(x,y) $=$ true iff x and y are adjacent (base(x) $+$ size(x) $=$ base(y) or base(y) $+$ size(y) $=$ base(x).
- OOB(x) $=$ true iff there exists an out-of-bound condition on memory buffer x.
- TGT(x) $=$ true iff memory cell x is an interesting target to overwrite.

Decide safety using Distance function

$$\forall x \neg \exists y : TGT(y) \wedge DOOB(x) > DIST(x, y)$$

- DIST(x,y) : $\mathbb{N}$ = | base(x) - base(y) |
- DOOB(x) : $\mathbb{N}$ is the maximum offset from x's base address that can be (over)written/read.
- TGT(y) = true iff chunk y is an interesting target to overwrite.

Automation challenges for Heap attacks

1. Do not confuse Logical and Spatial Heap semantics (Shape Analysis vs. Layout Analysis)
   - *Heap Models for Exploit Systems* (Vanegue, Langsec 2015)
   - *Automated Heap Layout Manipulation for Exploitation* (Heelan et al. to appear in Usenix Security 2018)
2. Decision of the ADJ(x,y) predicate is too approximate in the abstract. Requires tracking heap bins finely.
3. ADJ(x,y) is not separable for each heap bin: two chunks belonging to different bins could still be adjacent.
4. Each heap allocator uses different rules for memory management.
5. Heap distance across executions monotonically grows with time (a problem for heap-heavy programs, such as browsers)

CLASS 4: Automate Exploit Chaining

- Five years ago: "Multi-interaction exploits" was already a problem in the AEGC 2013
- Exploit Chaining is one of the main techniques used in real exploits today.
- Examples of Exploits Chain: Pinkie Pie Pwnium 2012 (chain of logic bugs and memory corruption to escape Chrome sandbox): Used pre-rendering feature to load Native client plug-in, from where triggered a buffer overflow in the GPU process, leading to impersonating a privileged renderer process via IPC squatting. From there, used an insecure Javascript-to-C++ interface to specify extension path to be loaded (impersonating the browser), and finally loaded an NPAPI plug-in running out of the sandbox. See "A Tale of Two Pwnies (Part 1)" by Obes and Schuh (Google Chromium blog)

Multi-interaction exploits (aka Exploit Chaining) leads

- As a matter of fact, little to no progress on automating chaining in last 5 years.
- *Weird Machines* characterize exploits as untrusted computations over a state machine.
- Problem: How to automate state creation on the weird machine?
- Formally: If a program is a function of type: $X \Rightarrow Y$, where X is an initial state leading to corrupted state Y then:

$$\exists Z : X \Rightarrow Z \wedge Z \Rightarrow Y$$

We dub this "The intermediate exploit state problem".

The Intermediate Exploit State problem

- There are whole chains of intermediates:

$$\exists Z_1, Z_2, ..., Z_n : X \Rightarrow Z_1 \wedge Z_1 \Rightarrow Z_2 \wedge ... \wedge Z_{n-1} \Rightarrow Z_n$$

- For each step i, is there a unique candidate Z_i? Not if state depends on control predicates (if/else/for conditions)
- Even for a single path, there may be multiple Z_i one could choose from. In particular, see "The Weird Machines in Proof Carrying Code" (Langsec 2013): characterize unaccounted intermediate steps in PCC.

CLASS 5: Information disclosure (ex: side-channel attacks)

Information disclosures (or "Info leak") has been used for at least 15 years in exploits.

- Direct info leaks (read uninitialized memory, OOB read, etc)
- Indirect info leaks (infer information from timing or other observable quantities)

In the last year, new hardware-based info leaks were publically released (Spectre, Meltdown, etc):

- Variant 1: Speculative bound check bypass (Jan 2018)
- Variant 2: Branch Target Buffer (Jan 2018)
- Variant 3: Rogue Data Cache Load (Jan 2018)
- Variant 4: Speculative Store Bypass (May 2018)

Ref: "Reading privileged memory with a side-channel" (by Jann Horn, Google P0)
Attack: Exploit speculative caching CPU feature for timing attacks.
Outcome: Attacker can predict bit values across privilege levels.

Spectre Variant 1 : a possible candidate for exploit automation

```
struct array { ulong len; uchar data[]; }

(...)
  struct array *arr1 = init_trusted_array(0x10);
  struct array *arr2 = init_trusted_array(0x400);
  ulong untrusted_offset = read_untrusted_len();
  if (untrusted_offset < arr1->len) {
    uchar value = arr1->data[untrusted_offset];
    uint  idx   = 0x200 + ((value & 1) * 0x100);
    if (idx < arr2->len)
       return (arr2->data[idx]);
  }
(...)
```

Insights

- Possible strategy: Assume CPU behavior, check programs for vulnerable code traits
- Interestingly: try to detect effects (cached state), not root cause (as usual).
- This is non-standard for static analysis (usually go after root cause by checking invariant, etc).
- Traditional black/grey/white-box fuzzers are blind to these properties.
- Checking such properties appears beyond compile-time analysis.
- Mitigations are already underway (ex: retpoline against Spectre Variant 2).
- Augmented static analysis or symbolic execution could be designed to keep track of cached states and speculative conditions (not trivial)

Another new problem: Automating Rowhammer-style attacks

Rowhammer is a hardware attack that can flip bits in memory with a probabilistic chance of success.
None of the discussed techniques would work to detect this:

- One need a probabilistic semantic to model such attacks.
- In spirit: Similar to brute-forcing a password: requires a lot of tries, success is aleatory.
- Possible approach: quantify attack success using techniques typically used by cryptographic security proofs.
- Possible outcome: prove hardware is secure with very high probability.
- Prediction: flaw will be fixed by design in next generation hardware.
- Counter-Prediction: probabilistic memory attacks are not going away, a framework is needed to study them.

Summing up

What is the next exciting autoresearch ahead?

	Done?	Some techniques	Few/No tech
Input Generation	X		
Exploit Specification		X	
State Space Management		X	
Primitive Composition			X
Information disclosure			X

Conclusion

Automated Exploit Generation is not yet solved in 2018.

Beware of folks telling you otherwise. People will try.

Questions?

Mail: julien.vanegue@gmail.com Twitter: @jvanegue