OptiROP: the art of hunting ROP gadgets

- Proposal for Blackhat USA 2013 -

Nguyen Anh Quynh <aquynh@gmail.com>

Abstract

Return-Oriented-Programming (ROP) is the fundamental technique to bypass the widely-used DEP-based exploitation mitigation. Unfortunately, available tools that can help to find ROP gadgets mainly rely on syntactic searching. This method proves to be in inefficient, time-consuming and makes the process of developing ROP-based shellcode pretty frustrated for exploitation writers.

This research attempts to solve the problem by introducing a tool named *OptiROP* that lets exploitation writers search for ROP gadgets with semantic queries. Combining sophisticated techniques such as code normalization, code optimization, code slicing, SMT solver and some creative heuristic searching methods, OptiROP is able to discover desired gadgets very quickly, with much less efforts. Our tool also provides the detail semantic meaning of each gadget found, so users can easily decide how to chain their gadgets for the final shellcode.

In case where no suitable gadget is found, OptiROP tries to pick and chain available gadgets to create a sequence of gadgets satisfying the input requirements. This significantly eases the hard job of shellcode writers, so they can focus their time on other tedious parts of the exploitation process.

This talk will entertain the audience with some live demo, so they can see how OptiROP generates gadgets in reality.

OptiROP supports input binary of all executable formats (PE/ELF/Macho) on x86 & x86_64 architectures. The tool will be released to the public after this talk for everybody to use.

1. Design and Implementation

This is only a draft paper describing OptiROP. The final paper will be providing more details on the techniques of our tool.

1.1 Semantic gadgets

1. Given an executable binary (we support all the format ELF/PE/Macho on 32-bit and 64-bit Intel platform at the moment), OptiROP gathers all the ROP gadgets from the executable segments of the input file. We simply use the classic method to find gadgets: OptiROP looks for the gadget tail (RET or JMP instructions), then walk back to some extents to gather meaningful sequences of code [3]. Each chunk of code like this is a candidate ROP gadget. All the ROP gadgets are saved and used as input of the next step.

2. OptiROP normalize all the ROP gadgets using LLVM IR [5]. We use LLVM framework to translate the machine code in each gadget into LLVM IR, and preserve the code semantics (Reference to Intel manual is a must at this step). At the output of this stage, for each ROP gadget we have a corresponding LLVM bitcode.

3. We run all the LLVM bitcode of gadgets through LLVM optimizers to remove junk code present after the normalized process [4]. This step optimizes the bitcode to be more compact, so it is easier/faster for the SMT solver to process the SMT queries deriving from the LLVM bitcode in the next stage.

4. We translate all the LLVM bitcode to SMT formula, so we have a formula for each original ROP gadget. We developed a LLVM pass for this process [6], and this pass analyzes the bitcode, then converts each LLVM instruction to Z3 formula [2] . All the original ROP gadgets and their SMT formulas are bundled together, and saved in the same place for searching later on.

5. Given a semantic query from user, the query is compiled to LLVM IR and its SMT formula is also generated like with the ROP gadgets. This is called *user formula* for brevity. Then one by one, on each SMT formula of the gathered ROP gadget (called *saved formula* for brevity), the SMT solver Z3 will be used to verify the equivalence of the saved formula and the user formula. The equivalent result confirms that this is the desired gadget, and the will be presented to user as searching result.

6. Semantic searching is much slower than syntactic searching, unfortunately. To improve the performance of the finding process, we have a simple observation: the most expensive operation in the whole system presented above is the equivalent verification step done by the SMT solver (Note that the steps from gathering gadgets to generating SMT formulas for them can be performed only once, and in advance, so we do not take that into account). For this reason, we should always use some quick and cheap heuristic verification methods first, and use SMT solver only if we cannot avoid that. In case we must run the solver, we try to simplify the formula as best as we can, so the solver only has to deal with simple equation, which can significantly improve the speedup the verification routine.

Guidance by these ideas, we use following methods, sorted by the order of execution.

(I) We quickly check for the registers that are modified at the output of each saved formula. Quick pattern matching can eliminate the formulas that do not clobber the same set of modified registers in the user formula. This simple trick immediately removes a lot of candidate gadgets in early stage, thus save a lot of time. Some evaluations told us that this step can remove up to 80% gadgets.

(II) We always want the shorter gadgets, so the next technique is to check to see if the verified gadget contains as its postfix some already-matched gadget. If so, we can discard this gadget right away. This trick can remove up to 10% gadgets in some experiments.

(III) All the saved formula actually includes a lot of "code" that are not related to the user query, thus can be removed without causing any impact to the equivalence checking. For example, a gadget might change EAX and EBX registers at the same time. But if the query only cares about EAX, all the code related to other registers (EBX and ESP in this case) can be removed from the formula. From this observation, we eliminate

all the "unrelated code" using the code slicing technique [7]. The code slicing technique will be run on the formula, and cleaned formula becomes much more compact, thus easier to be processed by the SMT solver. This trick alone can amazingly speed up the performance of the verification process up to 300% in some of our experiments.

(IV) We can avoid repeatedly verifying the slices by caching all the verified slice, and see if the same slice is already checked before running it through the SMT solver. Our experiments show that quite a lot of slices are shared between gadgets, and this trick can helps us cut the number of times running SMT solver by half in many cases.

(V) We observed that in some cases, the already-present logical constraint in the saved formula is also what we need to prove the equivalence. A prominent example is the time when we want to find the gadgets that move ESP register up by some concrete distance (like ESP += 0x20). The constraint on ESP is always present there in gadgets due to the RET instruction present at the end of them. So rather than proving the equivalence, we can simply append above constraint (ESP += 20) to the saved formula, and verify its validity. This method avoids the high expense of the *Exists* quantifier of equivalence verification, therefore much faster. Experiments showed that this trick can improve the performance by over 650% when applied.

1.2 Chain gadgets

Sometimes, OptiROP cannot find any suitable gadget from available gadgets. In this case, OptiROP can pick and chain some gadgets to have a sequence of gadgets performing desired request.

(I) OptiROP classifies the gadgets according to their semantics. We did this by generating the SMT formulas for all the gadgets, then verify them against a group of gadget types such as: copying concrete value to registers, copy register to register, load/store register from/to memory, arithmetic operations and execution branches.

With each gadget, OptiROP also saves information on the regiters being modified, and impact to memory (like stack). This is useful to chain the gadgets in the next step.

(II) Upon the user request, we try to combine the gadgets following on some rules proposed by [8]. The information on clobbered registers of each gadget is useful to match the gadgets so they are not incompatible.

(III) The candidate sequence of gadgets will be verified against the user request. This is done similar to how we check for the semantic equivalence of saved formula and user formula explained in the section 1.1. The matched gadgets will be present to user as result.

OptiROP has been implemented in Python and C++, which took about 10,000 lines of code. Attached in the appendix is a screen log of some OptiROP sessions, so the readers can see how OptiROP works in reality.

References

[1] LLVM project, http://www.llvm.org
[2] Z3 solver, http://z3.codeplex.com

[3] Shacham, H et all. “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”. ACM CCS 2007.
[4] http://llvm.org/docs/Passes.html
[5] http://llvm.org/docs/LangRef.html
[6] http://llvm.org/docs/WritingAnLLVMPass.html
[7] Weiser, M. "Program slicing". Proceedings of the 5th International Conference on Software Engineering, March 1981.
[8] Homescu, A et all. “Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming”. Usenix WOOT 2012.

Appendix

Below is a screen log copied from a session of OptiROP. All the text following '#' on some lines are comments from the author, but not present in the real output.

```
quynh@laptop$ ./optirop
Welcome to OptiROP version 1.0. By Nguyen Anh Quynh, 2012-2013
Type 'help' for quick introduction on the commands of the tool

> help
List of commands:
- help: quick help on OptiROP commands
- gen <file> [max-insn]: Generate gadgets for input binary
- load <file>: Load already-gathered gadgets of a particular binary
- constraint <condition>: List of constrainsts of the ROP gadgets to be
found later
- preserve <registers>: List of registers to be unmodified at the
output of the ROP gadgets
- badchars <chars>: List of bad characters that should not be present
in the output gadgets
- reset: Reset (clear) all the constraints, preserves and bad chars
- search [max-insn]: Start searching for ROP gadgets. The limit of
number of instructions is max-insn (unlimited by default)

> gen ./bin-linux-32/libc-2.15.so 4
This takes a while, please wait .... done!
Total gadgets: 30424
Non-duplicated gadgets: 4161
Time: 10m 26s

> constraint EAX = 0
OK, set constraint to EAX = 0

> search
Searching ... done.
```

```
   2df2e: xor eax, eax; mov esi, [esp+0x18]; add esp, 0x1c; ret
     Modified registers: eax, esi, ZF, CF, SF, OF, AF
     esp += 0x20

   2ec8b: xor eax, eax; pop esi; ret
     Modified registers: eax, esi, ZF, CF, SF, OF, AF

   2f0df: xor eax, eax; ret
     Modified registers: eax, ZF, CF, SF, OF, AF
     esp += 0x4

   341fa: xor eax, eax; add esp, 0x8; ret
     Modified registers: eax, ZF, CF, SF, OF, AF
     esp += 0xc

   565d0: xor eax, eax; add esp, 0xc; ret
     Modified registers: eax, ZF, CF, SF, OF, AF
     esp += 0x10

   69936: xor eax, eax; add esp, 0x1c; ret
     Modified registers: eax, ZF, CF, SF, OF, AF
     esp += 0x20

   7ec09: xor eax, eax; pop ebx; ret
     Modified registers: eax, ebx, ZF, CF, SF, OF, AF
     esp += 0x8

   83485: xor edx, edx; mov eax, edx; ret
     Modified registers: eax, edx, ZF, CF, SF, OF, AF
     esp += 0x4

  14868b: mov edi, 0x0; lea eax, [edi]; pop edi; ret
     Modified registers: eax, edi
     esp += 0x8

Gadgets sent through SMT solver: 932
Gadgets found: 9

> preserve ESI, EDX
OK, preserve registers ESI, EDX

> search
Searching ... done

   2f0df: xor eax, eax; ret
```

```
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x4

   341fa: xor eax, eax; add esp, 0x8; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0xc

   565d0: xor eax, eax; add esp, 0xc; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x10

   69936: xor eax, eax; add esp, 0x1c; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x20

   7ec09: xor eax, eax; pop ebx; ret
      Modified registers: eax, ebx, ZF, CF, SF, OF, AF
      esp += 0x8

  14868b: mov edi, 0x0; lea eax, [edi]; pop edi; ret
      Modified registers: eax, edi
      esp += 0x8

Gadgets sent through SMT solver: 787
Gadgets found: 6

> constraint EAX = 0; CF = 0
OK, constraint EAX = 0; CF = 0

> search
Searching ... done

   2f0df: xor eax, eax; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x4

   341fa: xor eax, eax; add esp, 0x8; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0xc

   565d0: xor eax, eax; add esp, 0xc; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x10

   69936: xor eax, eax; add esp, 0x1c; ret
```

```
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x20

   7ec09: xor eax, eax; pop ebx; ret
      Modified registers: eax, ebx, ZF, CF, SF, OF, AF
      esp += 0x8

Gadgets sent through SMT solver: 787
Gadgets found: 5

> reset
OK, reset all the constraints and preserve to None

> constraint EAX = EDX
OK, set constraint to EAX = EDX

> search
Searching ... done

   2f279: mov eax, edx; ret
      Modified registers: eax

   2fcdb: mov eax, edx; pop edi; pop ebp; ret
      Modified registers: eax, edi, ebp
      esp += 0xc

   55d54: mov eax, edx; add esp, 0x7c; ret
      Modified registers: eax, ZF, CF, SF, OF, AF
      esp += 0x80

   6c8d2: mov eax, edx; pop esi; ret
      Modified registers: eax, esi
      esp += 0x8

   7d30e: xchg edx, eax; add [eax], eax; add bh, dh; ret 0x3
      Modified registers: eax, ebx, edx, ZF, CF, SF, OF, AF
      esp += 0x7

   8a1a4: lea eax, [edx]; pop edi; ret
      Modified registers: eax, edi
      esp += 0x8

   8a2e4: lea eax, [edx]; ret
      Modified registers: eax
      esp += 0x4
```

```
   e87a4: mov eax, edx; add esp, 0x3c; ret
     Modified registers: eax, ZF, CF, SF, OF, AF
     esp += 0x40

Gadgets sent through SMT: 294
Gadgets found: 8

> constraint EAX = 12
Searching ... cannot find any gadget!
Trying to chain other gadgets to meet the requirement ... done
Chain following gadgets.

   # stack-top = 12
   1930e: pop edx; ret                  # EDX = 12
   8a2e4: lea eax, [edx]; ret           # EAX = EDX

     Modified registers: eax, edx

> constraint ECX = ESP
Searching ... cannot find any gadget!
Trying to chain other gadgets to meet the requirement ... done
Chain following gadgets.

    763d: imul ecx, [esi], 0x0; ret 0x0    # set ECX = 0
  b8d07: add ecx, esp; ret                 # ECX == ESP + 0 = ESP
     Modified registers: ecx, edx, ZF, CF, SF, OF, AF

> constraint ECX = [EDX]
Searching ... cannot find any gadget!
Trying to chain other gadgets to meet the requirement ... done
Chain following gadgets.

   2f279: mov eax, [edx]; ret               # EAX == [EDX]
    763d: imul ecx, [esi], 0x0; ret 0x0     # set ECX = 0
   c9cdf: xor ecx, eax                      # ECX == EAX
     Modified registers: ecx, edx, ZF, CF, SF, OF, AF
```