A stack model for symbolic buffer overflow exploitability analysis
Extended Abstract

Gustavo Grieco*
University of Grenoble
VERIMAG
Grenoble, France
Gustavo.Grieco@imag.fr

Laurent Mounier
University of Grenoble
VERIMAG
Grenoble, France
Laurent.Mounier@imag.fr

Marie-Laure Potet
University of Grenoble
VERIMAG
Grenoble, France
Marie-Laure.Potet@imag.fr

Sanjay Rawat†
University of Grenoble
LIG
Grenoble, France
Sanjay.Rawat@imag.fr

I. Introduction

Vulnerability analysis aims to detect programming flaws inside software code in order to prevent their exploitation by external attackers, for instance by control-flow hijacking. One of the most challenging issues in vulnerability analysis is being able to distinguish between *exploitable* and *non-exploitable* flaws. In this work we propose a symbolic approach to assess the exploitability level of a path leading to a flaw. This approach operates on (disassembled) binary code and starts with the identification of "dangerous paths", i.e., paths containing some patterns that depend on inputs [1]. Then, we produce the corresponding path conditions augmented by symbolic constraints dedicated to exploitability.

Predicates for buffer-overflow exploitability have been initially proposed in [2], [3], [4]. The novelty of our approach is to use a fully symbolic execution that requires to define an abstract memory model tractable for solvers. [2], [3], [4] use concretization to optimize read and write memory addresses, sacrificing completeness. The memory model must be accurate enough in order to capture exploitability: for instance buffer overflow exploits rely on the fact that objects lie adjacent to each others in memory and it is possible to *write* to one object by *overflowing* the other object [5]. Our memory model is very close to the one proposed in [6], used for Value Set Analysis, and relies on the notion of memory-regions. The advantages is twofold: memory regions are small enough to be tractable by solvers, and they fit well with reverse-engineering practices.

II. Exploitability formula generation

A. General Approach

We present *Symbolic Exploit Assistant* (SEA) a tool that implements the proposed approach. Symbolic exploitability generation is based on four steps:

1) The first input is a binary code which is disassembled (using IDA Pro) and then translated into the REIL intermediate representation (using BinNavi) [7]. REIL representation relies on a simple and limited instruction set, without implicit side effects, which makes it easier to analyze and process.
2) A second input is a symbolic trace σ defined as a sequence of REIL instructions from input functions to a potentially exploitable statement.
3) A formula π_{reach} is then produced from σ. This formula is a (simplified) path-condition giving the set of necessary input constraints to execute the control flow associated to σ.
4) Finally, we generate $\pi_{exploit}$, the set of necessary conditions that make σ exploitable (see section II-B for an example). If $\pi_{reach} \wedge \pi_{exploit}$ is satisfiable, then σ is an exploitable path.

Path-constraint generation from individual REIL instructions is rather straightforward. Since REIL does not handle native instructions (e.g. FPU, GPU) we only use the quantifier-free finite-precision bit-vector arithmetic theory provided by the SMT-LIB standard [8]. However, for scalability issues, we need to reduce the set of constraints produced. This is achieved by performing some backward slicing [9] to track the memory locations that influence (targeted statement (using classical trace dependency analysis). As a result, only constraints on these memory locations are included in π_{reach} and $\pi_{exploit}$.

B. Buffer-overflow exploitability

Figure 1 recalls how data are stored in the execution stack. In particular, the register `ebp` (on x86 architectures) points to the base address of the current function frame. Local variables are addressed through negative offsets from `ebp`, the *return address* is stored at address `ebp` + 4, and actual parameters are stored at addresses `ebp` + n with $n \geq 8$.

A classical way to hijack the program flow to a given address $addr_{shell}$ (the shellcode) is to overwrite the return address $addr_{ret}$, which can be done when a memory write occurs. Hence, we focus on specific vulnerable REIL instructions, such as the following one:

```
1: stm val, , addr      /* Mem[adr] := val */
```

Partially funded by the BINSEC project (ANR-12-INSE-002-01)
Partially funded by ITEA2 DIAMONDS project.

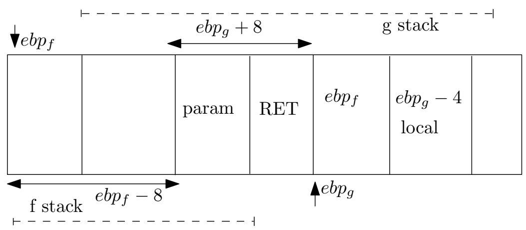

Figure 1. Execution stack for two functions f (caller) and g (callee)

From such an instruction, SEA extracts two sets of constraints:

- π_{reach} allowing to reach instruction l of σ,
- $\pi_{exploit} = C_{val} \cup C_{addr}$, with C_{val} (resp. C_{addr}) the set of input-dependent constraints enforcing $val = addr_{ret}$ (resp. $addr = addr_{shell}$).

C. A stack memory model

Here we define a stack memory model which enforces the locality of some variables (i.e. some local variables are next to others) allowing us to discover a wide range of exploits related with pointer misuse.

The memory model adopted by SEA produces a local array for every stack frame instance (denoted by $s_{f.i}$ where i is an index for each instance). These arrays are constrained by local read and write operations.

When a call is performed, a new stack frame is created with constraints identifying the bytes already pushed in the old stack frame as parameters. For example, let f and g be two functions where f (with frame index i) pushes a 32-bit value into the stack at $\text{ebp}_f - 8$ before calling g (with frame index j). The constraints needed to relate $s_{f.i}$ and $s_{g.j}$ are:

$$\begin{aligned} s_{f.i}[\text{ebp}_f - 8] &= s_{g.j}[\text{ebp}_g + 8] \\ s_{f.i}[\text{ebp}_f - 9] &= s_{g.j}[\text{ebp}_g + 7] \\ &\dots \end{aligned}$$

D. An illustrating example

We present a small example with an obvious vulnerability: it allows an attacker to manipulate the address and the content of a memory write. The steps 1) and 2) of our approach are already performed and the resulting REIL code (SSA form is omitted for clarity) is the following:

```
00: call
01: add   eax, ebx, t0   // t0 = eax + ebx
02: bisz  t0,      , zf  // t0 = 0 ? zf=1 : zf=0
03: jcc   zf,  1   , 15  // zf≠0 ? jmp 15
15: add   ebp, ecx, t1   // t1 = ebp+ecx
16: stm   eax,     , t1  // Mem[t1] := eax
..
99: ret
```

In this example `eax`, `ebx` and `ecx` are controlled by the attacker. Before reaching a memory write instruction (16), a conditional jump is performed (1–3) and a memory address is computed (15). The goal in this example is to overwrite the return address of the current stack frame with the value "0xdeadbeef" using the instruction 16. Eventually, the function reaches a return instruction (99), i.e., a jump to the address we selected. This requires that no write instructions change the return address. For path conditions, SEA outputs:

$$\begin{aligned} \pi_{reach} &= \{\text{zf} \neq 0,\ \ ite(t_0 = 0, \text{zf} = 1, \text{zf} = 0),\ \ t_0 = \text{eax} + \text{ebx}\} \\ C_{val} &= \{\text{eax} = 0xdeadbeef\} \\ C_{addr} &= \{t_1 = \text{ebp} + 4, t_1 = \text{ebp} + \text{ecx}\} \end{aligned}$$

Solving $\pi_{reach} \cup \pi_{exploit}$, the values of the user-controlled registers for obtaining the expected memory write obtained by Z3 are:

$$\boxed{\text{eax} = 0xdeadbeef, \quad \text{ebx} = 0x21524111, \quad \text{ecx} = 0x4}$$

III. Conclusion

We implemented a POC of SEA using z3py, the official Python interface of Z3 [10]. We plan to strengthen exploitability constraints in order to take into account some extra conditions (e.g. preconditions defined in [5]). We also plan to target other form of exploits, such as used after free. To do that the proposed memory model must be extended.

Acknowledgment

Sincere thanks to anonymous reviewers for useful remarks.

References

[1] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables," in *Proceedings of SERE 2012*. Washington DC, USA: IEEE, 2012, pp. 177–186.

[2] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," Master's thesis, 2009.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *NDSS*, 2011.

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symp. S&P*, 2012, pp. 380–394.

[5] M. Bishop, S. Engle, D. Howard, and S. Whalen, "A taxonomy of buffer overflow characteristics," *IEEE Trans. on Dependable and Secure Computing*, vol. 9, no. 3, pp. 305 –317, may-june 2012.

[6] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010.

[7] T. Dullien and P. Sebastian, "REIL : A platform-independent intermediate representation of disassembled code for static c ode analysis," in *Proceedings of CanSecWest*, 2009.

[8] C. Barrett and al., "The SMT-LIB Standard: Version 2.0," Tech. Rep., 2010.

[9] M. Weiser, "Program slicing," in *Proceedings of the 5th ICSE '81*. IEEE Press, 1981, pp. 439–449.

[10] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.