Guide Me To Exploit: Assisted ROP Exploit Generation for ActionScript Virtual Machine

UNC CHARLOTTE
College of Computing and Informatics
Department of Software and Information Systems

Fadi Yilmaz[1], Meera Sridhar[2], Wontae Choi[3]
[1]fadiyilmaz@ybu.edu.tr, [2]msridhar@uncc.edu, [3]wtchoi.kr@gmail.com

Motivation

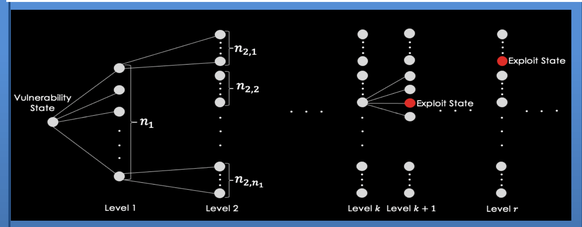
Monitoring the execution of exploit scripts is crucial

Writing exploit for each vulnerability requires intense human-effort

Discovering unorthodox methods to exploit vulnerabilities

Needing a tool that produces exploit scripts in an autonomous fashion

Automated Exploit Generation

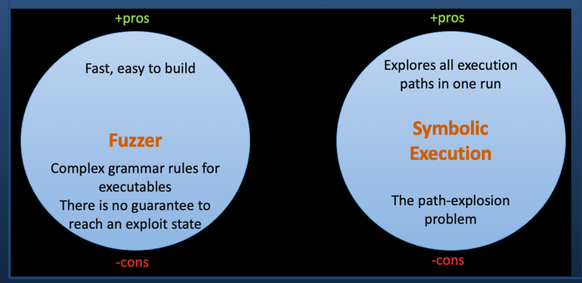


The challenge of determining the exploitability of given vulnerability [1]

Explore all execution paths that follow the vulnerability state [2]

Hope one the execution paths reaches one of exploit states

Utilize a fuzzer [3] or a symbolic executor [4]



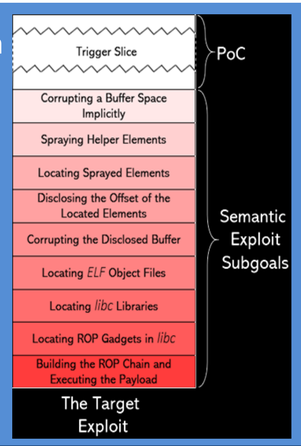
Intuition Behind Target Exploit Generation

The exploit pattern

Malicious activity must be performed under the radar

Deconstructing the exploit scripts into smaller subgoals

Subgoals are defined by the security experts



Exploit Subgoals

Consist of a search space and an invariant

Set of instructions that can contribute to achieve the corresponding exploit subgoal

The test that checks whether the corresponding exploit subgoal is achieved

Methodology

Focusing on synthesizing exploit scripts that achieve an exploit subgoal in sequence instead of focusing on generating the entire exploit at once

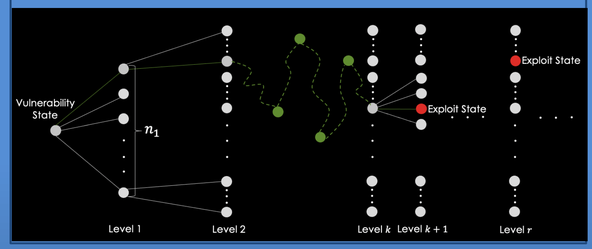
Synthesizing a distinct executable for every permutation of instructions given in the search space

Methodology – Cont.

Testing synthesized executables with the given invariant

Focusing on synthesizing executables for the next exploit subgoal after the current exploit subgoal is achieved

Stitching up executables that achieve an exploit subgoal to obtain the final exploit script



Optimization Techniques

Exploit Deconstruction
Allows us to define exploit subgoals
Enables us to solve same problem with smaller instances (~10^{45})

Stack Simulation
Disqualifies candidate slices that perform illegal stack operation (98.78%)

Instruction Tiling
Creates bigger and more meaningful instruction chains to be used (~10^{13})

Feedback from the AVM
Allows us to detect error-raising instruction sequences and disqualifies candidate slices that contain such instruction sequence (58%)

Experimental Results

Our tool generates the exploit script in less than 15 minutes

Our tool exploits a use-after-free vulnerability CVE-2015-5119, which is one of the most commonly exploited vulnerabilities in 2016

Our tool synthesized more than 650 million executables to reach the target exploit state

Exploit Subgoal	Number of Generated Candidate Slices	Synthesizing Time (s)
Corrupting a Buffer Space Implicitly	2,396,744	9.35
Spraying Helper Elements	19,173,95	55.90
Locating Sprayed Elements	37,448	1.72
Disclosing the Offset of the Located Elements	55,345,75	138.26
Corrupting the Disclosed Buffer	4,793,488	17.03
Locating `ELF` Object Files	19,173,95	57.12
Locating `libc` Libraries	55,345,757	138.05
Locating Executable Segment	76,695,80	199.78
Locating Gadgets and Building the ROP Chain	435,848,049	240.92
		Total: 858.13 (14m 18.13s)

Future Work

Conducting a user study to measure

- How much human interaction is required
- What level of human expertise is required
- How much effort does our tool save for a seasoned developer

Key References

[1] [Younis et al. SQJ'16] Awad Younis, Yashwant K Malaiya, and Indrajit Ray. 2016. Assessing vulnerability exploitability risk using software properties. Software Quality Journal 24, 1 (2016), 159–202.
[2] [Avgerinos et al. NDSS'11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In Proceedings of The Network and Distributed System Security Symposium (NDSS).
[3] [Miller et al. ACM'90] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33, 12 (1990), 32–44.
[4]v[King et al. ACM'76] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.