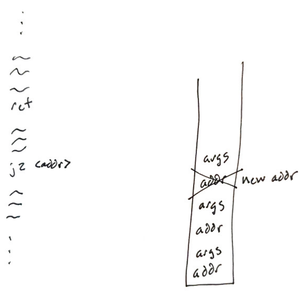
Using Planning to Construct Code Reuse Attacks in Obfuscated Programs

Naiqian Zhang[1], Daroc Alden[1], Dongpeng Xu[1], Shuai Wang[2], Trent Jaeger[3], and Wheeler Ruml[1]

1 University of New Hampshire
2 The Hong Kong University of Science and Technology
3 PennState

Code Reuse Attacks

1. calling subroutines pushes return address on stack
2. given write vulnerability, **write new return addresses**
3. control flow goes wherever we want!



Previous Work

ROPGadget: predefined gadget patterns and chain templates
fast but fails if any part of pattern is absent

Angrop: 'semantic patterns' via symbolic execution
still uses chain templates
can't build primitives from multiple gadgets

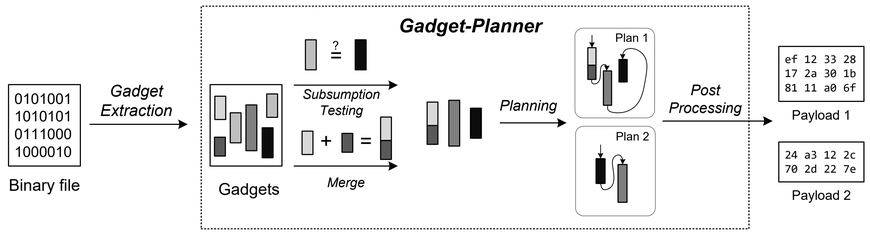
SGC: state as QF_ABV formulas, find chains via SMT
contemporaneous with our early work
$\sim$ planning as SAT, uses subset, tries various chain lengths
can't handle direct or conditional jumps

This Work: Gadget-Planner

- find and characterize all gadgets
- use partial-order planning to arrange attack
- more robust: **finds more attacks**
- security result: especially in obfuscated binaries

full paper: Zhang et al, "No Free Lunch: On the Increased Code Reuse Attack Surface of Obfuscated Programs," *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-23)*, 2023

Gadget-Planner



Gadget code

```
inc rax
sub rdx, rbx
jnz 0x1234
ret
```

Gadget 'action'

starting address: `0x4321`
length: `16`
pre: `rbx == rdx`
post: $\texttt{rax}_{\text{after}} == \texttt{rax}_{\text{before}} + 1$
modified: `rax, rdx`

pre and post via symbolic execution
Z3 + 'pointer' type
subsumption: $(pre_2 \rightarrow pre_1) \wedge (post_1 = post_2)$

Goal for `execve` attack

```
rax = 57
rdi = "/path/to/mal.exe"
rsi = Pointer(0)
rdx = Pointer(0)
```

Partial-Order Planning

- partial order = small search space
- lifted: we don't care about most of the state
- not obvious how to use constraints to derive heuristic
- don't care about attack length
- mainly want to find any attack ASAP

heuristics:

- prefer fewer open preconditions
- prefer fewer constraints and fewer symbolic variables
- prefer simpler gadgets

Experimental Results

	Goal	mean attcks (new w/ obf) ropgg	angrop	SGC	ours
none	`execve`	1	1	6	**8**
	`mprotect`	-	1	5	**6**
	`mmap`	-	1	5	**6**
	Total	1	3	16	**20**
llvm	`execve`	1 (0)	1 (0)	6 (0)	**15** (7)
	`mprotect`	-	1 (0)	5 (0)	**10** (4)
	`mmap`	-	1 (0)	5 (0)	**12** (6)
	Total	1 (0)	3 (0)	16 (0)	**37** (17)
tigress	`execve`	1 (0)	1 (0)	6 (0)	**16** (8)
	`mprotect`	-	1 (0)	5 (0)	**16** (10)
	`mmap`	-	1 (0)	5 (0)	**9** (3)
	Total	1 (0)	3 (0)	16 (0)	**41** (21)

more attacks!

obfuscated code more exploitable!

	|gadgt|	|atck|	Ret	IJ	DJ	CJ
ROPGadget	2.1	12.6	100%	0%	0%	0%
Angrop	2.3	13.8	100%	0%	0%	0%
SGC	5.8	23.2	68%	32%	0%	0%
ours	6.7	33.5	38%	10%	12%	40%

gadgets & chains longer, **more diverse**

stage	time (sec)	mem (GB)
`Gadget Extraction`	2,223	4.16
`Subsumption Testing`	2,104	5.73
`Planning`	1,771	8.43
Total	6,098	

planning is not the most expensive part!

Conclusions

- a new application of planning in security
- simple case of program synthesis
- **works! yay, planning!**